SpEL Dynamic Array Access in APEX Rules

This guide provides comprehensive examples and best practices for accessing dynamic arrays in Spring Expression Language (SpEL) expressions within APEX rules.

Table of Contents

- Basic Array Access
- Safe Navigation
- Dynamic Index Access
- Array Bounds Checking
- Collection Operations
- Financial Data Examples
- Advanced Patterns
- Error-Safe Patterns
- Best Practices
- Common Pitfalls

Basic Array Access

Bracket Notation Syntax

SpEL offers three different ways to access nested properties and arrays. Each has specific use cases:

Option 1: Pure Bracket Notation

```yaml
condition: "#trade['otcTrade']['otcLeg'][0]['stbRuleName'] != null"
```

When to use:

- ✅ **Dynamic property names**: When property names come from variables
- ✅ **Special characters**: Property names with spaces, hyphens, or special chars
- ✅ **Consistent syntax**: Same syntax for properties and array indices
- ✅ **Runtime flexibility**: Property names determined at execution time

Example with dynamic properties:

```yaml
# Property name comes from a variable
condition: "#trade[#propertyName]['otcLeg'][0]['stbRuleName'] != null"
# Where #propertyName might be 'otcTrade', 'equityTrade', etc.
```

Option 2: Mixed Notation (Recommended)

```yaml
condition: "#trade.otcTrade.otcLeg[0]['stbRuleName'] != null"
```

When to use:

- ✅ **Best of both worlds**: Readable for known properties, flexible for dynamic ones
- ✅ **Known structure**: Use dot notation for fixed property names
- ✅ **Dynamic parts**: Use brackets only where needed (arrays, dynamic properties)
- ✅ **Most common**: Typical pattern in real-world APEX rules

Why this is often preferred:

```
# Clear structure: object.property.array[index]['dynamicProperty']
condition: "#portfolio.positions.trades[#tradeIndex]['customField'] != null"
#          ^^^^^^^^^ ^^^^^^^^^ ^^^^^^ ^^^^^^^^^^^ ^^^^^^^^^^^^^
#          object    property  array   dynamic      dynamic
#                                     index        property
```

Option 3: Pure Dot Notation

```
condition: "#trade.otcTrade.otcLeg[0].stbRuleName != null"
```

When to use:

- ✅ **Fixed structure**: All property names are known at compile time
- ✅ **Simple objects**: No special characters in property names
- ✅ **Readability**: Most readable when structure is predictable
- ✅ **IDE support**: Better autocomplete and syntax highlighting

Limitations:

```
# ❌ Won't work if property names have special characters
condition: "#trade.otc-trade.otc-leg[0].stb-rule-name != null"  # INVALID

# ❌ Won't work with dynamic property names
condition: "#trade.#propertyName.otcLeg[0].stbRuleName != null"  # INVALID
```

Comparison Summary

Aspect	**Bracket** `['prop']`	**Mixed** `.prop['dynamic']`	**Dot** `.prop`
Readability	⚠️ Verbose	✅ **Best balance**	✅ Clean
Dynamic Properties	✅ Full support	✅ Partial support	❌ No support
Special Characters	✅ Handles all	✅ In brackets only	❌ Limited
Performance	⚠️ Slightly slower	✅ **Optimal**	✅ Fastest
Maintenance	⚠️ More typing	✅ **Recommended**	✅ Simple

Real-World Examples

```
# Financial trading scenario - mixed notation (recommended)
condition: "#trade.otcTrade.legs[#legIndex].notionalAmount > 1000000"
#          ^^^^^ ^^^^^^^ ^^^^ ^^^^^^^^^^ ^^^^^^^^^^^^^^^
#          obj   prop    arr  dynamic    known property

# Configuration-driven scenario - bracket notation required
condition: "#config[#ruleType][#jurisdiction]['mandatoryFields'].contains('LEI')"
#          ^^^^^^^ ^^^^^^^^^ ^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^
#          obj     dynamic   dynamic      known property

# Simple object access - dot notation preferred
condition: "#customer.profile.riskRating == 'HIGH'"
#          ^^^^^^^^^ ^^^^^^^ ^^^^^^^^^^
#          obj       prop    prop
```

Array Element Access

```
# Access first element
condition: "#positions[0].instrumentId != null"

# Access specific index
condition: "#trades[2].tradeId != null"

# Access last element (if size is known)
condition: "#items[#items.size() - 1].status == 'COMPLETE'"
```

Safe Navigation

Always use safe navigation (`?.`) to prevent null pointer exceptions:

```
# Safe array access - prevents errors if any level is null
condition: "#trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"

# Safe access with bracket notation throughout
condition: "#trade?.['otcTrade']?.['otcLeg']?.[0]?.['stbRuleName'] != null"

# Mixed safe navigation
condition: "#portfolio?.positions?.[0]?.trades?.size() > 0"
```

Dynamic Index Access

For truly dynamic array access, there are two main patterns:

Pattern 1: Search-Based Access (Most Common)

When you need to find an array element by condition, but don't know which index will match:

```
# Find first leg where legType equals 'FLOATING' - position unknown
condition: "#trade.otcTrade.otcLeg.^[legType == 'FLOATING']?.stbRuleName != null"

# Find first position with specific instrument type - index unknown
```

```yaml
condition: "#portfolio.positions.^[instrumentType == 'BOND']?.quantity > 0"

# Find pay leg in swap trade - could be at any array position
expression: "#trade.legs.^[payReceive == 'PAY']?.notionalAmount"
```

Why this is most common:

- ✅ **Business Logic Driven**: Find elements based on business criteria
- ✅ **Position Independent**: Works regardless of array ordering
- ✅ **Robust**: Handles data structure variations
- ✅ **Self-Documenting**: Clear what you're searching for

Pattern 2: Variable Index Access

For truly dynamic array access where the index itself is variable:

How Variable Index Resolution Works

Understanding the step-by-step process of how SpEL resolves variable array indices:

Expression Resolution Logic

Expression: `#trade.otcTrade.otcLeg[#trade.selectedLegIndex].stbRuleName != null`

Step-by-Step Resolution:

1. `#trade` → Resolves to the `trade` HashMap from context data
2. `#trade.selectedLegIndex` → Resolves to the integer value (e.g., `1`)
3. `#trade.otcTrade` → Resolves to the `otcTrade` HashMap
4. `#trade.otcTrade.otcLeg` → Resolves to the `List<Map<String, Object>>` array
5. `#trade.otcTrade.otcLeg[#trade.selectedLegIndex]` → Becomes `otcLeg[1]` → Resolves to element at index 1
6. `#trade.otcTrade.otcLeg[#trade.selectedLegIndex].stbRuleName` → Resolves to the property value
7. **Final evaluation:** `propertyValue != null` → `true` or `false`

Data Structure Example

```yaml
# Test data structure:
trade:
  selectedLegIndex: 1           # ← Dynamic index value
  otcTrade:
    otcLeg:
      - stbRuleName: "RULE_A"    # ← Index 0
      - stbRuleName: "RULE_B"    # ← Index 1 ← Selected by selectedLegIndex
      - stbRuleName: "RULE_C"    # ← Index 2

# SpEL Resolution Process:
#trade.selectedLegIndex         → 1
#trade.otcTrade.otcLeg[1]       → { stbRuleName: "RULE_B" }
#trade.otcTrade.otcLeg[1].stbRuleName → "RULE_B"
```

Why Dynamic Indexing Works

1. **SpEL evaluates expressions inside brackets first** - `[#trade.selectedLegIndex]` is evaluated before array access
2. **The result becomes the array index** - The integer `1` becomes the literal index
3. **Array access happens with resolved index** - `otcLeg[1]` accesses the second element

4. **Property access continues normally** - `.stbRuleName` accesses the property of the resolved element

Runtime Index Selection

The index is determined at runtime based on data values:

```
# Different selectedLegIndex values produce different results:

# If selectedLegIndex = 0
otcLeg[#trade.selectedLegIndex] → otcLeg[0] → "RULE_A"

# If selectedLegIndex = 1
otcLeg[#trade.selectedLegIndex] → otcLeg[1] → "RULE_B"

# If selectedLegIndex = 2
otcLeg[#trade.selectedLegIndex] → otcLeg[2] → "RULE_C"
```

Dynamic Search-Based Index Resolution

Common Scenario: You need to find an array element by searching for a matching condition, but you don't know which index will match.

Search Pattern: Find First Matching Element

```
# Find the first leg where legType equals 'FLOATING'
expression: "#trade.otcTrade.otcLeg.^[legType == 'FLOATING']"
# Resolution: Searches array, returns first element where legType == 'FLOATING'

# Find the first leg with a specific currency
expression: "#trade.otcTrade.otcLeg.^[currency == 'USD']"
# Resolution: Returns first USD leg, regardless of its array position
```

Search Pattern: Find Last Matching Element

```
# Find the last leg where notionalAmount > 1000000
expression: "#trade.otcTrade.otcLeg.$[notionalAmount > 1000000]"
# Resolution: Searches array backwards, returns last high-value leg
```

Search Pattern: Find All Matching Elements

```
# Find all legs where maturityDate is not null
expression: "#trade.otcTrade.otcLeg.?[maturityDate != null]"
# Resolution: Returns array of all legs that have maturity dates
```

Real-World Search Examples

```
# Financial trading scenarios - search-based access
condition: "#trade.otcTrade.otcLeg.^[legType == 'FIXED']?.stbRuleName != null"
# Find first fixed leg and check if it has a rule name

condition: "#trade.otcTrade.otcLeg.^[payReceive == 'PAY']?.notionalAmount > 0"
# Find first pay leg and validate positive notional
```

```yaml
expression: "#trade.otcTrade.otcLeg.?[currency == 'EUR'].size()"
# Count how many legs are in EUR

condition: "#portfolio.positions.^[instrumentType == 'BOND']?.maturityDate != null"
# Find first bond position and check maturity date
```

Search vs Index Comparison

Approach	When to Use	Example
Known Index	Index is predetermined	`otcLeg[#selectedLegIndex]`
Search First	Find first match	`otcLeg.^[legType == 'FLOATING']`
Search Last	Find last match	`otcLeg.$[status == 'ACTIVE']`
Search All	Find all matches	`otcLeg.?[currency == 'USD']`

Data Structure Example for Search

```yaml
# Test data structure - unknown which leg is floating:
trade:
  otcTrade:
    otcLeg:
      - legType: "FIXED"         # ← Index 0 - not what we want
        stbRuleName: "RULE_A"
        currency: "USD"
      - legType: "FLOATING"      # ← Index 1 - this is what we're searching for!
        stbRuleName: "RULE_B"    # ← This is the value we want to extract
        currency: "USD"
      - legType: "FIXED"         # ← Index 2 - not what we want
        stbRuleName: "RULE_C"
        currency: "EUR"

# Search Resolution Process:
#trade.otcTrade.otcLeg.^[legType == 'FLOATING']           → { legType: "FLOATING", stbRuleName: "RULE_B", currency: "USD"
#trade.otcTrade.otcLeg.^[legType == 'FLOATING'].stbRuleName → "RULE_B"
```

Safe Search Patterns

```yaml
# ✅ SAFE: Check if search found a result
condition: "#trade.otcTrade.otcLeg.^[legType == 'FLOATING'] != null"
condition: "#trade.otcTrade.otcLeg.^[legType == 'FLOATING']?.stbRuleName != null"

# ✅ SAFE: Provide fallback if no match found
expression: "#trade.otcTrade.otcLeg.^[legType == 'FLOATING']?.stbRuleName ?: 'DEFAULT_RULE'"

# ❌ UNSAFE: Assumes search will always find a match
condition: "#trade.otcTrade.otcLeg.^[legType == 'FLOATING'].stbRuleName != null"
```

Variable Index Examples

```yaml
# Using a variable index
condition: "#trade.otcTrade.otcLeg[#legIndex].stbRuleName != null"
# Resolution: #legIndex → 2, otcLeg[2] → third element
```

```
# Dynamic index from another field
condition: "#trade.otcTrade.otcLeg[#trade.selectedLegIndex].stbRuleName != null"
# Resolution: #trade.selectedLegIndex → 1, otcLeg[1] → second element

# Safe dynamic index access
condition: "#trade?.otcTrade?.otcLeg?.size() > #legIndex && #trade.otcTrade.otcLeg[#legIndex]?.stbRuleName != null"
# Resolution: Checks bounds (size > index) before accessing element

# Dynamic index with calculation
condition: "#items[#currentIndex + 1]?.status != null"
# Resolution: #currentIndex + 1 → 0 + 1 = 1, items[1] → second element
```

Real-World Variable Index Applications

```
# Financial trading scenarios
condition: "#trade.legs[#trade.payLegIndex].currency == 'USD'"
condition: "#portfolio.positions[#portfolio.primaryPositionIndex].quantity > 0"
condition: "#basket.instruments[#basket.selectedInstrumentIndex].maturityDate != null"

# Risk management scenarios
condition: "#riskLimits.thresholds[#riskProfile.severityLevel].maxExposure > #currentExposure"
condition: "#counterparties[#trade.counterpartyIndex].creditRating in {'AAA', 'AA+'}"

# Regulatory reporting scenarios
condition: "#reportingRules[#jurisdiction.ruleSetIndex].mandatoryFields.contains('LEI')"
condition: "#complianceChecks[#trade.productType.checkIndex].required == true"
```

Array Bounds Checking

Always check array bounds before accessing elements:

```
# Check array exists and has elements before accessing
condition: "#trade?.otcTrade?.otcLeg != null && #trade.otcTrade.otcLeg.size() > 0 && #trade.otcTrade.otcLeg[0].stbRuleNam

# Check specific index exists
condition: "#trade?.otcTrade?.otcLeg != null && #trade.otcTrade.otcLeg.size() > 2 && #trade.otcTrade.otcLeg[2].stbRuleNam

# More concise with safe navigation
condition: "#trade?.otcTrade?.otcLeg?.size() > 0 && #trade.otcTrade.otcLeg[0]?.stbRuleName != null"

# Check minimum array size
condition: "#positions?.size() >= 3 && #positions[2].quantity > 0"
```

Collection Operations

SpEL provides powerful collection operations for dynamic arrays:

Filtering Operations

```
# Find first element matching condition
condition: "#trade?.otcTrade?.otcLeg?.^[stbRuleName == 'SPECIFIC_RULE'] != null"

# Find last element matching condition
```

```yaml
condition: "#trade?.otcTrade?.otcLeg?.$[stbRuleName != null] != null"

# Check if any element matches condition
condition: "#trade?.otcTrade?.otcLeg?.?[stbRuleName != null].size() > 0"

# Filter elements by multiple conditions
condition: "#positions?.?[quantity > 0 && instrumentType == 'EQUITY'].size() > 0"
```

Projection Operations

```yaml
# Get all stbRuleNames from the array
expression: "#trade?.otcTrade?.otcLeg?.![stbRuleName]"

# Get all quantities from positions
expression: "#positions?.![quantity]"

# Project nested properties
expression: "#trades?.![legs?.![ruleName]]"

# Project with null safety
expression: "#items?.![name != null ? name : 'UNKNOWN']"
```

Aggregation Operations

```yaml
# Count elements matching condition
expression: "#trade?.otcTrade?.otcLeg?.?[stbRuleName != null].size()"

# Sum all quantities
expression: "#positions?.![quantity].sum()"

# Get maximum value
expression: "#trades?.![notionalAmount].max()"

# Get minimum value
expression: "#trades?.![notionalAmount].min()"
```

Financial Data Examples

Real-world examples for OTC trade processing:

```yaml
rules:
  # Check if any leg has a specific rule
  - id: "otc-leg-rule-check"
    name: "OTC Leg Rule Validation"
    condition: "#trade?.otcTrade?.otcLeg?.?[stbRuleName == 'MARGIN_RULE'].size() > 0"
    message: "At least one leg must have margin rule"
    severity: "ERROR"

  # Validate all legs have required fields
  - id: "all-legs-complete"
    name: "All Legs Complete Validation"
    condition: "#trade?.otcTrade?.otcLeg?.?[stbRuleName == null || stbRuleName.trim().isEmpty()].size() == 0"
    message: "All legs must have stbRuleName specified"
    severity: "ERROR"
```

```yaml
  # Access specific leg by position with safety
  - id: "first-leg-validation"
    name: "First Leg Validation"
    condition: "#trade?.otcTrade?.otcLeg?.size() > 0 && #trade.otcTrade.otcLeg[0]?.stbRuleName?.matches('[A-Z_]+')"
    message: "First leg must have valid rule name format"
    severity: "WARNING"

  # Dynamic leg access based on trade type
  - id: "dynamic-leg-access"
    name: "Dynamic Leg Access"
    condition: "#trade?.tradeType == 'SWAP' && #trade?.otcTrade?.otcLeg?.size() >= 2 && #trade.otcTrade.otcLeg[1]?.stbRul
    message: "Swap trades must have second leg with rule name"
    severity: "ERROR"

  # Portfolio position validation
  - id: "portfolio-position-check"
    name: "Portfolio Position Validation"
    condition: "#portfolio?.positions?.?[quantity <= 0 || instrumentId == null].size() == 0"
    message: "All positions must have positive quantity and valid instrument ID"
    severity: "ERROR"

  # Risk limit validation across positions
  - id: "risk-limit-check"
    name: "Risk Limit Validation"
    condition: "#portfolio?.positions?.![notionalValue].sum() <= #riskLimits.maxPortfolioValue"
    message: "Portfolio value exceeds risk limits"
    severity: "ERROR"
```

Advanced Patterns

For more complex scenarios, we show both complex and simpler maintainable alternatives:

Complex vs Simple: Nested Array Access

```yaml
# ❌ COMPLEX: Deep nested access in one expression
condition: "#portfolio?.positions?.[#positionIndex]?.trades?.[#tradeIndex]?.legs?.[0]?.ruleName != null"

# ✅ SIMPLE: Break into multiple readable steps
condition: "#portfolio?.positions?.size() > #positionIndex"
condition: "#portfolio.positions[#positionIndex]?.trades?.size() > #tradeIndex"
condition: "#portfolio.positions[#positionIndex].trades[#tradeIndex]?.legs?.size() > 0"
condition: "#portfolio.positions[#positionIndex].trades[#tradeIndex].legs[0]?.ruleName != null"
```

Complex vs Simple: Multi-Level Projections

```yaml
# ❌ COMPLEX: Extract all rule names from nested structure in one expression
expression: "#portfolio?.positions?.![trades?.![legs?.![ruleName]]].flatten()"

# ✅ SIMPLE: Process each level separately for clarity
condition: "#portfolio?.positions?.size() > 0"
expression: "#portfolio.positions.![trades]"  # Get all trades from all positions
expression: "#allTrades.![legs]"              # Get all legs from all trades
expression: "#allLegs.![ruleName]"            # Get all rule names from all legs
```

Complex vs Simple: Conditional Logic

```yaml
# ❌ COMPLEX: Ternary operator with nested array operations
condition: >
  #trade?.structure == 'SIMPLE' ?
    (#trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null) :
    (#trade?.otcTrade?.otcLeg?.?[stbRuleName != null].size() == #trade.otcTrade.otcLeg.size())

# ✅ SIMPLE: Separate rules for different trade structures
# Rule 1: Handle simple trades
condition: "#trade?.structure == 'SIMPLE' && #trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"

# Rule 2: Handle complex trades - all legs must have rule names
condition: "#trade?.structure != 'SIMPLE'"
condition: "#trade?.otcTrade?.otcLeg?.size() > 0"
condition: "#trade.otcTrade.otcLeg.?[stbRuleName != null].size() == #trade.otcTrade.otcLeg.size()"
```

Complex vs Simple: Multi-Condition Filtering

```yaml
# ❌ COMPLEX: Multiple filters and projection in one expression
expression: "#trades?.?[notionalAmount > 1000000 && counterparty?.rating in {'AAA', 'AA+', 'AA'}]?.![tradeId]"

# ✅ SIMPLE: Step-by-step filtering for better readability and debugging
condition: "#trades?.size() > 0"
expression: "#trades.?[notionalAmount > 1000000]"                  # Step 1: Filter by amount
expression: "#highValueTrades.?[counterparty?.rating != null]"       # Step 2: Has rating
expression: "#ratedTrades.?[counterparty.rating in {'AAA', 'AA+', 'AA'}]"  # Step 3: High rating
expression: "#qualifiedTrades.![tradeId]"                           # Step 4: Extract IDs
```

Complex vs Simple: Dynamic Property Access

```yaml
# ❌ COMPLEX: Ternary with dynamic property access
expression: "#data[#propertyName] != null ? #data[#propertyName] : #data['defaultProperty']"

# ✅ SIMPLE: Explicit null checking with clear fallback logic
condition: "#data[#propertyName] != null"
expression: "#data[#propertyName]"        # Use when condition passes
expression: "#data['defaultProperty']"    # Use as fallback in separate rule
```

When to Use Complex vs Simple Patterns

✅ Use Complex Patterns When:

- **Performance Critical**: Single expression is significantly faster than multiple rule evaluations
- **Atomic Operations**: You need all-or-nothing logic that can't be split
- **Mathematical Calculations**: Complex financial formulas that must execute as one unit
- **Experienced Team**: All team members are comfortable with advanced SpEL

✅ Use Simple Patterns When:

- **Team Readability**: New team members need to understand the logic quickly
- **Debugging Required**: You need to trace through logic step-by-step
- **Frequent Changes**: Business logic changes often and needs easy modification
- **Testing Focus**: You want to test each logical step independently

🎯 Recommended Approach

1. **Start Simple**: Begin with readable, step-by-step expressions
2. **Optimize Later**: Combine into complex expressions only if performance requires it
3. **Document Complex**: Always add comments explaining complex expressions
4. **Test Both**: Ensure complex and simple versions produce identical results

Error-Safe Patterns

Comprehensive Safe Dynamic Array Access

```yaml
# Complete safe dynamic array access
condition: >
  #trade != null &&
  #trade.containsKey('otcTrade') &&
  #trade.otcTrade != null &&
  #trade.otcTrade.containsKey('otcLeg') &&
  #trade.otcTrade.otcLeg != null &&
  #trade.otcTrade.otcLeg instanceof T(java.util.List) &&
  #trade.otcTrade.otcLeg.size() > 0 &&
  #trade.otcTrade.otcLeg[0] != null &&
  #trade.otcTrade.otcLeg[0].containsKey('stbRuleName') &&
  #trade.otcTrade.otcLeg[0].stbRuleName != null

# More concise version using safe navigation
condition: "#trade?.otcTrade?.otcLeg?.size() > 0 && #trade.otcTrade.otcLeg[0]?.stbRuleName != null"
```

Type-Safe Array Access

```yaml
# Verify array type before access
condition: "#data.items instanceof T(java.util.List) && #data.items.size() > 0"

# Safe casting with type check
expression: "#data.items instanceof T(java.util.List) ? #data.items[0] : null"

# Multiple type checks
condition: "#trade?.legs instanceof T(java.util.List) && #trade.legs.size() > 0 && #trade.legs[0] instanceof T(java.util.
```

Null-Safe Collection Operations

```yaml
# Safe filtering with null checks
expression: "#items?.?[# != null && #.status != null && #.status == 'ACTIVE'] ?: {}"

# Safe projection with fallbacks
expression: "#positions?.![quantity != null ? quantity : 0] ?: {}"

# Safe aggregation
expression: "#values?.![# != null ? # : 0].sum() ?: 0"
```

Best Practices

1. Prioritize Readability Over Cleverness

```yaml
# ✅ PREFERRED - Clear, step-by-step logic
condition: "#trade?.structure == 'SIMPLE'"
condition: "#trade?.otcTrade?.otcLeg?.size() > 0"
condition: "#trade.otcTrade.otcLeg[0]?.stbRuleName != null"

# ❌ AVOID - Clever but hard to debug
condition: "#trade?.structure == 'SIMPLE' && #trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"
```

2. Always Use Safe Navigation

```yaml
# ✅ Good - safe navigation prevents NPE
condition: "#trade?.otcTrade?.otcLeg?.size() > 0"

# ❌ Bad - can throw NullPointerException
condition: "#trade.otcTrade.otcLeg.size() > 0"
```

3. Check Array Bounds

```yaml
# ✅ Good - bounds checking
condition: "#items?.size() > 2 && #items[2]?.status == 'ACTIVE'"

# ❌ Bad - no bounds checking
condition: "#items[2].status == 'ACTIVE'"
```

4. Break Complex Logic Into Steps

```yaml
# ✅ PREFERRED - Multiple simple rules
# Rule 1: Check high value
condition: "#trade?.notionalAmount > 1000000"
# Rule 2: Check counterparty rating
condition: "#trade?.counterparty?.rating in {'AAA', 'AA+', 'AA'}"
# Rule 3: Extract trade ID
expression: "#trade.tradeId"

# ❌ AVOID - One complex expression
expression: "#trades?.?[notionalAmount > 1000000 && counterparty?.rating in {'AAA', 'AA+', 'AA'}]?.![tradeId]"
```

3. Use Collection Operations for Filtering

```yaml
# ✅ Good - use collection operations
condition: "#trades?.?[status == 'PENDING'].size() > 0"

# ❌ Less efficient - manual iteration would be needed
```

4. Validate Data Types

```yaml
# ✅ Good - type validation
condition: "#data.items instanceof T(java.util.List) && #data.items.size() > 0"

# ❌ Risky - assumes type without checking
```

```
condition: "#data.items.size() > 0"
```

5. Use Meaningful Variable Names

```
# ✅ Good - clear variable names
condition: "#currentLegIndex < #trade.otcTrade.otcLeg.size()"

# ❌ Less clear - generic names
condition: "#i < #trade.otcTrade.otcLeg.size()"
```

Practical Real-World Examples

Trade Processing: Complex vs Simple

```
# SCENARIO: Validate a multi-leg derivative trade

# ❌ COMPLEX: Everything in one expression
condition: "#trade?.legs?.size() > 1 && #trade.legs.?[notional > 0 && currency != null && maturityDate != null].size() ==

# ✅ SIMPLE: Break into logical steps
condition: "#trade?.legs?.size() > 1"                          # Multi-leg trade
condition: "#trade.legs.?[notional > 0].size() == #trade.legs.size()"     # All legs have positive notional
condition: "#trade.legs.?[currency != null].size() == #trade.legs.size()" # All legs have currency
condition: "#trade.legs.?[maturityDate != null].size() == #trade.legs.size()" # All legs have maturity
condition: "#trade.legs.![currency].toSet().size() == 1"       # All legs same currency
```

Risk Management: Complex vs Simple

```
# SCENARIO: Check if portfolio exceeds risk limits

# ❌ COMPLEX: Nested calculations in one expression
condition: "#portfolio?.positions?.![notionalAmount * riskWeight].sum() > #riskLimits?.maxExposure && #portfolio.position

# ✅ SIMPLE: Step-by-step risk calculation
expression: "#portfolio.positions.![notionalAmount * riskWeight]"  # Calculate weighted exposures
expression: "#weightedExposures.sum()"                             # Total exposure
condition: "#totalExposure > #riskLimits?.maxExposure"            # Check limit breach
expression: "#portfolio.positions.?[counterparty?.rating not in {'AAA', 'AA+'}]" # Non-prime positions
expression: "#nonPrimePositions.![notionalAmount].sum()"          # Non-prime exposure
condition: "#nonPrimeExposure > #riskLimits?.maxNonPrimeExposure" # Check non-prime limit
```

Regulatory Reporting: Complex vs Simple

```
# SCENARIO: Extract required fields for regulatory report

# ❌ COMPLEX: Multi-level extraction in one expression
expression: "#trades?.?[reportingRequired == true]?.![{tradeId: tradeId, counterparty: counterparty?.lei, notional: legs?

# ✅ SIMPLE: Clear field extraction steps
condition: "#trades?.?[reportingRequired == true].size() > 0"    # Has reportable trades
```

```yaml
expression: "#reportableTrades.![tradeId]"                     # Extract trade IDs
expression: "#reportableTrades.![counterparty?.lei]"           # Extract LEIs
expression: "#reportableTrades.![legs?.![notionalAmount].sum()]" # Calculate notionals
expression: "#reportableTrades.![legs?.[0]?.currency]"         # Extract currencies
```

Common Pitfalls

Syntax Errors

```yaml
# ❌ Wrong - incorrect bracket syntax
condition: "#trade.['otcTrade'].['otcLeg'][0].['stbRuleName'] != null"

# ✅ Correct - proper bracket syntax
condition: "#trade['otcTrade']['otcLeg'][0]['stbRuleName'] != null"
```

Null Pointer Exceptions

```yaml
# ❌ Wrong - can cause NullPointerException
condition: "#trade.otcTrade.otcLeg[0].stbRuleName != null"

# ✅ Correct - safe navigation
condition: "#trade?.otcTrade?.otcLeg?.[0]?.stbRuleName != null"
```

Array Bounds Errors

```yaml
# ❌ Wrong - no bounds checking
condition: "#trade.otcTrade.otcLeg[5].stbRuleName != null"

# ✅ Correct - bounds checking
condition: "#trade?.otcTrade?.otcLeg?.size() > 5 && #trade.otcTrade.otcLeg[5]?.stbRuleName != null"
```

Type Assumptions

```yaml
# ❌ Wrong - assumes array type
condition: "#data.items[0].name != null"

# ✅ Correct - validates type first
condition: "#data.items instanceof T(java.util.List) && #data.items.size() > 0 && #data.items[0]?.name != null"
```

Performance Issues

```yaml
# ❌ Inefficient - repeated expensive operations
condition: "#expensiveCalculation()[0] != null && #expensiveCalculation()[0].value > 100"

# ✅ Efficient - calculate once, store in variable
condition: "#result = #expensiveCalculation(); #result?.size() > 0 && #result[0]?.value > 100"
```

Summary

Key Takeaways: Simple vs Complex SpEL

🎯 **Primary Recommendation: Start Simple, Optimize Later**

Dynamic array access in APEX SpEL expressions requires careful attention to:

- **Readability First**: Use simple, step-by-step expressions that team members can easily understand and debug
- **Safe navigation** to prevent null pointer exceptions (`?.` operator)
- **Bounds checking** to avoid array index errors (`size() > index`)
- **Type validation** to ensure data structure assumptions are correct
- **Collection operations** for efficient filtering and projection when needed
- **Performance considerations** to avoid repeated expensive operations

The Simple-First Approach

1. ✅ **Start with readable expressions** - Break complex logic into multiple simple rules
2. ✅ **Test each step independently** - Easier debugging and validation
3. ✅ **Document business intent** - Clear rule names and descriptions
4. ✅ **Optimize only when needed** - Combine into complex expressions only for performance
5. ✅ **Keep complex examples** - Show what's possible, but prefer simple patterns

When You See Complex SpEL

- **Understand it**: Complex examples show SpEL's full capabilities
- **Question it**: Ask "Can this be simpler and more maintainable?"
- **Refactor it**: Break into steps unless performance demands complexity
- **Document it**: If complexity is necessary, explain why

Remember: The most elegant code is often the simplest code that clearly expresses business intent.

By following these patterns and prioritizing simplicity, you can create robust, maintainable SpEL expressions that handle dynamic arrays safely and efficiently in your APEX rules.