ISO/IEC/JTC 1/SC 22/WG 23 N1469

18 Feb 2025

Notes on Java Switch Statement, Erhard P and Stephen M

- Here are summarized findings for the new switch expressions and statements for basic types and their enhanced versions for reference types:

  In general, violating `switch` rules that result in compiler error(s) prohibits vulnerabilities from occurring since the program simply will not run. However, due to the long and confusing evolution of the Java `switch` statement and expression, I agree that a moderate sampling of the java `switch` permutations deserves representation in the document even if it is somewhat tutorial in nature. Section 5 may be a good home for some of this material.

  Note: Section 5 of the Java document does not have the detail that the Python document does (just an observation).

- The manual for Java 23 states in 14.11.2 : For compatibility reasons, switch statements that are not enhanced switch statements are not required to be exhaustive. … If the switch statement is an enhanced switch statement, then it must be exhaustive.

  Below is some additional information that may be useful if we want to define "**enhanced** `switch` statement" (potentially in Section 5):
  "
  Enhanced `switch` Statement:
  Introduced in Java 12 and refined in subsequent versions, the enhanced `switch` statement provides a more concise and flexible way to handle multiple conditional branches. It offers several key improvements over the *traditional* `switch` statement.

  - Arrow Syntax:
    The enhanced switch uses `->` instead of `case` and `break`, making the code more readable and less prone to errors caused by accidental fall-through.
  - Multiple `case` Labels:
    It allows multiple values to be associated with a single `case`, simplifying code when several inputs should result in the same action.
  - `switch` Expressions:
    The enhanced `switch` can be used as an expression, returning a value directly, which is assigned to a variable or used in another expression.
  - `yield` Keyword:
    When used as an expression, the `yield` keyword is employed to return a value from a case block, replacing the need for `break` with a value.

  "
- For switch expressions, it states in 15.28.1: It is a compile-time error if a switch expression is not exhaustive.

  Perhaps we should define "exhaustive" in Section 5 using14.11.1.1 as a guide:

"

- There is a `default` label associated with the `switch` block.
- There is a case `null, default` label associated with the `switch` block.
- The set containing all the `case` constants and `case` patterns appearing in an unguarded `case` label (collectively known as `case` elements) associated with the `switch` block is non-empty and covers the type of the selector expression e.

"

- So, a general claim for completeness checks would be incorrect.

  Agree.

- An issue that surprised me along the way was that the new switch rules and guards can be used in old-style case statements that need breaks to not proceed into the next case, but that such continuations are (necessarily) allowed only into branches that do not include variable declarations in their switch rules. The language rules to prevent that are a dilly.

  This surprised me as well. Below is a simple example that illustrates how the old (:) and new (->) nomenclature is valid for `switch` **statements**:

```
----------------------------------------
int value = 5;
switch (value){
    case 1:
        System.out.println("Value is 1");
    case 2:
        System.out.println("Value is 2");
    case 5:
        System.out.println("Value is 5");
    default:
        System.out.println("No Matches");
}
```

<u>Output</u>:

```
Value is 5
No Matches
----------------------------------------
int value = 5;
switch (value){
    case 1 -> System.out.println("Value is 1");
    case 2, 3 -> System.out.println("Value is 2"); // This is OK
    case 5 -> System.out.println("Value is 5");
    default -> System.out.println("No Matches ");
}
```

<u>Output</u>:

```
Value is 5
```

==

It's worth noting that multiple consecutive `case` statements ***can*** share a variable declaration in certain scenarios, as show in the example below, as long as fall-through is not restricted (via. `break` or `->`):

```
int choice = 1;
switch (choice) {
    case 1:
    case 2:{
        String message1 = "First and Second case";
        System.out.println(message1);
        break;
    }
    case 3: {
        String message2 = "Third case";
        System.out.println(message2);
        break;
    }
    default: {
        System.out.println("Default case");
    }
}
```

Output:

```
First and Second case
```

In the above example, `message1` is exclusively accessible to the `case 1` and `case 2` *shared* code block, and `message2` is only accessible within the `case 3` code block.

- And, as Sean already pointed out, a later switch rule must not be dominated (in a Java-defined sense) by an earlier one. It was interesting to see the pages of rules that prevent undecidability because of guards and complication by other interactions, e.g., of unboxing.

  The following additional context may help if we need it:

  "

  In Java's switch statements with pattern matching, case dominance refers to the order in which case labels are evaluated. When multiple case labels could potentially match the selector expression, the case that appears earlier in the switch block takes precedence. This means that if a case label "dominates" another, the dominated case will never be executed.

  Dominance is determined based on the type and structure of the patterns in the case labels. A case label with a more general pattern dominates a case label with a more specific pattern. For instance, `case Number n` dominates `case Integer i` because every `Integer` is also a `Number`.

  The compiler enforces dominance rules to prevent unreachable code. If a case label is dominated by a preceding case, a compile-time error will occur. This ensures that the switch statement is well-defined and behaves as expected. **To avoid dominance issues, case labels should be ordered from most specific to most general**.

  "

The following findings from tests are consistent with the manual.

- Incompleteness checks switches for Enum types: Agree

  switch expression: static error message naming the missing case

```
enum Status {
    OPEN,
    IN_PROGRESS,
    CLOSED
}

Status currentStatus = Status.OPEN;

String result = switch (currentStatus) {
    case OPEN -> "Status is Open";
    //case IN_PROGRESS -> "Status is In Progress";
    case CLOSED -> "Status is Closed";
};

System.out.println(result);
```

  Output:

```
The switch expression does not cover all input values
```

  switch statement: static warning, no runtime checks, Fall-Thru semantics for missing cases!

  There were no incompleteness warnings for the example below on my platform (using the default settings, but this could be configured if desired)

  Product Version: Apache NetBeans IDE 24
  Java: 22.0.1; Java HotSpot(TM) 64-Bit Server VM 22.0.1+8-16

```
        enum Status {
            OPEN,
            IN_PROGRESS,
            CLOSED
        }

        Status currentStatus = Status.OPEN;

        switch (currentStatus) {
            case OPEN:
                System.out.println("Status is Open");
                break;
//          case IN_PROGRESS:
//              System.out.println("Status is In Progress");
//              break;
            case CLOSED:
                System.out.println("Status is Closed");
                break;
        }
```

  Output:

```
Status is Open
```

- Incompleteness checks for int types: Agree

  switch expression: static error message asking for "default" case
  switch statement: NO warning, no runtime checks, Fall-Thru semantics for missing cases!

- Incompleteness checks for classes: Agree

  both switch statements and switch expressions produce a static error message, asking for "default" case

- Incompleteness checks for subclasses of sealed classes (where the set of subclasses is final and known)

  both switch statements and switch expressions check complete coverage (and, in case of error, ask for "default" rather than naming the missing class)

Next actions: Read the existing text with all this in mind....

==

Another interesting scenario uses the `continue` statement within the `switch`. The results are not overly surprising, but the behavior is something to be aware of and could lead to unexpected results if not implemented correctly:

```
for (int i = 0; i < 4; i++) {
    switch (i) {
        case 1:
            System.out.println("Case 1");
            continue; // This continue applies to the for loop
        case 2:
            System.out.println("Case 2");
            break;
        default:
            System.out.println("Default case");
    }
    System.out.println("After switch");
}
```

OUTPUT:
```
Default case
After switch
Case 1
Case 2
After switch
Default case
After switch
```