**ISO/IEC/JTC 1/SC 22 WG 23 N1463**

 **5 Feb 2025**

**Switch Statement and Expression analysis by S. McDonagh**

**Case 1:**

Action: WEDNESDAY deleted in the switch statement

Expectation: Static error on incompleteness in both routines. (If not, see Cases 2a and 2b, but it’s presumed that this case is handled as described.)

Results:

* **(No deletions)**

Expression Result:

WEDNESDAY is a weekday : true

Statement Result:

WEDNESDAY is a weekday : false

FALL THRU!!!

* **(Statement w/ deletion)** No error when WEDNESDAY is deleted and control switches to the end of the switch control block. In this scenario, an incorrect value (false) is returned so this behavior could result in a vulnerability.
* **(Expression w/ deletion)** COMPILATION ERROR when WEDNESDAY is deleted unless default handler is included.

import java.time.\*;

public class Case1 {

 public static boolean swStmt(DayOfWeek X) { // Switch Statement

 boolean Res = false;

 switch(X) {

 case MONDAY, TUESDAY, // multiple cases can be combined

 WEDNESDAY, // OMISSION causes no error and control

 // transfers to the end of the switch block.

 THURSDAY, FRIDAY -> Res = true;

 case SATURDAY, SUNDAY -> Res = false;

 }

 return Res;

 }

 public static boolean swExpr(DayOfWeek X) { // Switch Expression

 return switch(X) {

 case MONDAY, TUESDAY, // multiple cases can be combined

 WEDNESDAY, // OMISSION causes compilation error except if default

 THURSDAY, FRIDAY -> true;

 case SATURDAY, SUNDAY -> false;

 //default -> throw new IllegalStateException("Invalid day: " + X);

 };

 }

 public static void main(String[] args) {

 DayOfWeek Testday = DayOfWeek.WEDNESDAY;

 try {

 System.out.println("Expression Result:");

 System.out.println(Testday + " is a weekday : " + swExpr(Testday));

 }

 catch (Exception E) { System.out.println(E.getClass().getName()); }

 try {

 System.out.println("Statement Result:");

 System.out.println(Testday + " is a weekday : " + swStmt(Testday));

 }

 catch (Exception E) { System.out.println(E.getClass().getName()); }

 if (Testday == DayOfWeek.WEDNESDAY)

 System.out.println("FALL THRU!!!");

 }

}

=========================================================================

**Case 2:**

Action: Use int in switch statement, but no default branch.

Expectation: A static error on incompleteness (insists on a default

branch)

Results:

* **(No deletions w/ default**s**)**

Expression Result:

Day Name: Thursday, is Day Number: 4

Statement Result:

Day Name: Thursday, is Day Number: 4

FALL THRU!!!

* **(Statement w/ no default)** No error with missing default statement and program control switches to the end of the switch control block. In this scenario, the correct value (Thursday) is returned.
* **(Expression w/ no default)** “COMPILATION ERROR, the switch expression does not cover all possible input values.”

public class Case2 {

 public static String swStmt(int X) { // Switch Statement

 String dayName = "";

 switch(X) {

 case 1:

 dayName = "Monday";

 break;

 case 2:

 dayName = "Tuesday";

 break;

 case 3:

 dayName = "Wednesday";

 break;

 case 4:

 dayName = "Thursday";

 break;

 case 5:

 dayName = "Friday";

 break;

 case 6:

 dayName = "Saturday";

 break;

 case 7:

 dayName = "Sunday";

 break;

 default: // OMISSION causes no error and control transfers

. dayName = "Invalid day"; // to the end of the switch block

 break;

 }

 return dayName;

 }

 public static String swExpr(int X) { // Switch Expression

 String dayName = switch (X) {

 case 1 -> "Monday";

 case 2 -> "Tuesday";

 case 3 -> "Wednesday";

 case 4 -> "Thursday";

 case 5 -> "Friday";

 case 6 -> "Saturday";

 case 7 -> "Sunday";

 default -> "Invalid day"; // OMISSION results in COMPILATION ERROR,

 // the switch expression does not cover

 // all possible input values

 };

 return dayName;

 }

 public static void main(String[] args) {

 int dayNumber = 4;

 try {

 System.out.println("Expression Result:");

 System.out.println("Day Name: " + swExpr(dayNumber) + ", is Day Number: " + dayNumber );

 }

 catch (Exception E) { System.out.println(E.getClass().getName()); }

 try {

 System.out.println("Statement Result:");

 System.out.println("Day Name: " + swStmt(dayNumber) + ", is Day Number: " + dayNumber );

 }

 catch (Exception E) { System.out.println(E.getClass().getName()); }

 if (dayNumber == 4)

 System.out.println("FALL THRU!!!");

 }

}

=========================================================================

**Case 2a**:

Action: Call with a value that hits a branch:

Expectation: Either succeeds (in which case, the claim of completeness checking is a bit

overstated) or gives an "incompleteness exception"

Results: Reference 2 (above)

=========================================================================

**Case 2b:**

Action: Call with a value not in the switch

Expectation: Fall-Through Exception, surely by the switch expression, but also on switch stmt.?

Results:

* **(Statement & Expression w/ default & Invalid Day)** No errors for either scenario! No apparent vulnerability risk for this scenario.

Expression Result:

Day Name: Invalid day, is Day Number: 8

Expression Result:

Day Name: Invalid day, is Day Number: 8

FALL THRU!!!

* **(Statement w/ no default & Invalid Day)** OMISSION causes no error and control transfer to the end of the switch block.
* **(Expression w/ no default & Invalid Day)** “COMPILATION ERROR, the switch expression does not cover all possible input values.”

public class Case2b {

 public static String swStmt(int X) {

 String dayName = "";

 switch(X) {

 case 1:

 dayName = "Monday";

 break;

 case 2:

 dayName = "Tuesday";

 break;

 case 3:

 dayName = "Wednesday";

 break;

 case 4:

 dayName = "Thursday";

 break;

 case 5:

 dayName = "Friday";

 break;

 case 6:

 dayName = "Saturday";

 break;

 case 7:

 dayName = "Sunday";

 break;

 default: // OMISSION causes no error and control transfer

 // to the end of the switch block.

 dayName = "Invalid day";

 break;

 }

 return dayName;

 }

 public static String swExpr(int X) {

 String dayName = switch (X) {

 case 1 -> "Monday";

 case 2 -> "Tuesday";

 case 3 -> "Wednesday";

 case 4 -> "Thursday";

 case 5 -> "Friday";

 case 6 -> "Saturday";

 case 7 -> "Sunday";

 default -> "Invalid day"; // OMISSION causes COMPILATION ERROR,

 // the switch expression does not cover

 // all possible input values

 };

 return dayName;

 }

 public static void main(String[] args) {

 int dayNumber = 8; **// Invalid day number**

 try {

 System.out.println("Expression Result:");

 System.out.println("Day Name: " + swExpr(dayNumber) + ", is Day Number: " + dayNumber );

 }

 catch (Exception E) { System.out.println(E.getClass().getName()); }

 try {

 System.out.println("Expression Result:");

 System.out.println("Day Name: " + swStmt(dayNumber) + ", is Day Number: " + dayNumber );

 }

 catch (Exception E) { System.out.println(E.getClass().getName()); }

 if (dayNumber == 8) {

 System.out.println("FALL THRU!!!");

 }

 }

}

=========================================================================

**Case 3:**

Action: Declare a module A with a class with three subclasses; Do a type

switch with all three of them, no default. Call with a variable of one

of the subclasses.

Expectation: as in 2 or 2a

Results:

* **(Statement & Expression w/ default)** No errors for either scenario! No apparent vulnerability risk for this scenario.

Statement Results:

Drawing a circle

Drawing a rectangle

Drawing a triangle

Expression Results:

Drawing a Circle

Drawing a Rectangle

Drawing a Triangle

package com.mycompany.case3;

class Shape {

 public void draw() {

 System.out.println("Drawing a generic shape");

 }

}

class Circle extends Shape {

 @Override

 public void draw() {

 System.out.println("Drawing a circle");

 }

}

class Rectangle extends Shape {

 @Override

 public void draw() {

 System.out.println("Drawing a rectangle");

 }

}

class Triangle extends Shape {

 @Override

 public void draw() {

 System.out.println("Drawing a triangle");

 }

}

public class Case3 {

 public static void main(String[] args) {

 Shape shape1 = new Circle();

 Shape shape2 = new Rectangle();

 Shape shape3 = new Triangle();

 System.out.println("Statement Results:");

 drawShapeStmt(shape1);

 drawShapeStmt(shape2);

 drawShapeStmt(shape3);

 System.out.println("Expression Results:");

 System.out.println(drawShapeExpr(shape1));

 System.out.println(drawShapeExpr(shape2));

 System.out.println(drawShapeExpr(shape3));

 }

 // Switch Statement

 static void drawShapeStmt(Shape shape) {

 switch (shape) {

 case Circle c -> c.draw();

 case Rectangle r -> r.draw();

 case Triangle t -> t.draw();

 default -> shape.draw(); // OMISSION results in COMPILATION ERROR, the switch expression

 // does not cover all possible input values

 }

 }

 // Switch Expression

 static String drawShapeExpr(Shape shape) {

 return switch (shape) {

 case Circle c -> "Drawing a Circle";

 case Rectangle r -> "Drawing a Rectangle";

 case Triangle t -> "Drawing a Triangle";

 default -> "Drawing a Generic Shape"; // OMISSION results in COMPILATION ERROR, the

 // switch expression does not cover all possible

 // input values

 };

 }

}

=========================================================================

=========================================================================

The **switch** **Statement**

Switch **Statement** with The Colon Notation

 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");

 }

Output:

Value is 5

No Matches

Caution: As shown in the above example, once the switch **statement** used with the colon notation finishes executing the block of code corresponding to the input constant, it moves down and falls-through to the next constant. To avoid that, you need to use a break to take the execution flow out of the switch construct.

Switch **Statement** With The Arrow(->) Notation

* **Multiple case labels are not allowed**

The arrow notation does not allow multiple case labels to be associated with the same action, unlike the colon notation.

The code below will not compile:

 **int** value = 5;

 **switch** (value){

 **case** 1 -> System.out.println("Value is 1");

 **case** 2 -> **case** 3 -> System.out.println("Value is 2");

*// COMPILATION ERROR : Illegal start of statement*

 **case** 5 -> System.out.println("Value is 5");

 **default** -> System.out.println("No Matches");

 }

However, this is an acceptable alternative:

 **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 ");

 }

* **Actions associated with a case label are limited**

You cannot use a group of statements in an arrow notation switch.
The code below will not compile:

 **int** value = 5;

 **switch** (value){

 **case** 1 ->

 System.out.println("Value is 1");

 System.out.println("This line will not compile");

*// COMPILATION ERROR : Case, default, or } expected Not allowed*

**case** 2, 3 -> System.out.println("Value is 2");

 **case** 5 -> System.out.println("Value is 5");

 **default** -> System.out.println("No Matches");

 }

You can only use one of the following actions:

1. An [expression statement](https://nkamphoa.com/expressions-statements-and-blocks-in-java):

 **case** 2, 3 -> System.out.println("Value is …");

1. A [block](https://nkamphoa.com/expressions-statements-and-blocks-in-java) of statements:

 **case** 1 ->{

 System.out.println("Value is 1");

 System.out.println("This line will compile");

 }

1. Throw an exception:

 **default** -> **throw** **new** IllegalArgumentException("Not valid");

* **The Arrow Switch Execution is Mutually Exclusive**

Unlike the colon notation switch, there is no need for a break statement. Once the execution of the statements associated with a case label has been completed, the switch construct also terminates with no fall-through.

**Example of The Arrow Notation Switch Statement with no break**

 int value = 5;

 switch (value){

 case 1 ->{

 System.out.println("Value is 1");

 System.out.println("This line will compile");

 }

 case 2, 3 -> System.out.println("Value is either 2 or 3");

 case 5 -> System.out.println("Value is 5");

 default -> throw new IllegalArgumentException("Not a valid");

 }

* **Using Strings as Case Constants in Switch Statements**

“Starting from **Java 7**, you can use the String data type in your switch constructs. However, there are a couple of things you should be aware of while switching on Strings.

* The Java compiler compares the string constants based on their hash values first(integer values), followed by an object equality (using the equals() method) to rule out any collision.
* Switching on strings is less efficient than switching on integers. Therefore, you must only switch on strings if the values are already of type String.
* The compiler must be able to determine the value of all the *case constants* at compile-time.”

**Correct Example of Switching on Strings**

 String color = "blue";

 **final** String YELLOW = "yellow";

 **switch** (color){

 **case** "red" -> System.out.println("Color is red");

 **case** "blue" -> System.out.println("Color is blue");

 **case** YELLOW -> System.out.println("Color is yellow");

 **default** -> **throw** **new** IllegalArgumentException("Not valid");

 }

**Incorrect Way of Switching on Strings**

The following example **will not compile** because the compiler cannot guarantee that the value of the case constants will not change at runtime.

 String color = "blue";

 String YELLOW = "yellow";

 String RED = **new** String("red");

 **switch** (color){

 **case** RED -> System.out.println("Color is red");

*// Compile-time error, RED is not a constant*

**case** "blue" -> System.out.println("Color is blue");

 **case** YELLOW -> System.out.println("Color is yellow");

*// Compile-time error, YELLOW is not a constant*

**default** -> **throw** **new** IllegalArgumentException("Not valid");

 }

=========================================================================

The **switch** **Expression**

The **Switch expression** has the same semantics as a switch statement except it returns a value. Just like switch statements, there are two forms of switch expressions: Colon notation and Arrow notation switch expressions.

* **The yield Statement**

In switch expressions, the yield statement plays a similar role as the  break in switch statements. The yield statement can only be used in switch expressions.

 yield expression;

Execution of the statement above will return the value of expression as the result of the switch expression.

* **The Switch Expression With The Colon ”:” Notation**

The switch expression with the colon notation is analogous to the switch statement with the colon notation with the difference that it returns a value (or throws an exception).

 dataType switchValue = **switch**(selector\_expression){

 **case** value1: statements\_value1;

 yield someValue1;

 **case** value2: statements\_value2;

 yield someValue2;

 ....

 **case** valueN: statements\_valueN;

 yield someValueN;

 **default**: statements\_default;

 yield someDefaultValue;

 }

* dataType is the data type of switch expression value
* yield is used to return a value to the switch expression.
* Just like the switch statement, if there is no yield at the end of the group of statements, the execution will fall through the next group, if any.
* The switch expression with the colon notation must be **exhaustive**, meaning that the case labels, and if necessary the default label, must cover **all values** of the selector expression type. **Failure to cover all values will result in a compile-time error.** The **default** label is usually used to make sure the switch expression is exhaustive.

Below is an example of the switch expression with colon “:” notation:

 **int** value = 5;

 **int** switchValue = **switch**(value){

 **case** 1:

 System.out.println("Value is 1");

 yield 1;

 **case** 2:

 System.out.println("Value is 2");

 yield 2;

 **case** 3,4:

 System.out.println("Value is 3 or 4");

 yield 3;

 **default**:

 System.out.println("Value not in range");

 yield 0;

 }; *//Don't forget the semicolon (;)*

 System.out.println(switchValue);

The switch expression return a value into a variable, you must add a semicolon ( after the closing curly brace (}).

* **The Switch Expression With The Arrow(->) Notation**

The switch expression with arrow notation is simply a switch statement with the arrow notation that returns a value or throws an exception. Its syntax is as follows:

 dataType switchValue = **switch**(selector\_expression){

 **case** value1 -> statements\_value1;

 yield someValue1;

 **case** value2 -> statements\_value2;

 yield someValue2;

 ....

 **case** valueN -> statements\_valueN;

 yield someValueN;

 **default** -> statements\_default;

 yield someDefaultValue;

 }

Note the following:

1. The execution of the switch rules in a switch expression is **mutually exclusive**, just like in switch statements. Once the action in the switch has completed its execution, the value is returned to the switch expression and the switch body terminates. There is no **fall-through**.
2. Unlike switch statements, the action body is not limited to expression statements. In addition to using a statement block, or throwing an exception, you can use **any type of expression**.

**case** 1 -> 1; //simple expression, must return a valid value directly

**case** 2 -> yield 2; //compiled-time error, yield is not allowed here

**case** 3 -> { //statement block

 System.out.println("Value is 3");

 yield 3;

 }

**case** 4 -> { //statement block

 yield 4; // Compiled-time error, yield must be the last

 // statement in the block

 System.out.println("Value is 4");

 }

**default** -> **throw** **new** IllegalArgumentException("Not valid");

1. The switch statement with the arrow notation must also be **exhaustive** and all possible selector values must be covered.

**Example of Switch Expression with the Arrow(->) Notation**

 **int** value = 3;

 **int** switchValue = **switch**(value){

 **case** 1 ->{

 System.out.println("Value is 1");

 yield 1;

 }

 **case** 2-> 2;

 **case** 3,4->{

 System.out.println("Value is 3 or 4");

 yield 3;

 }

 **default**->

 **throw** **new** IllegalArgumentException("Not a valid value");

 };

 System.out.println(switchValue);

Output:

Value is 3 or 4

3