An Advanced version of Switch Case

CS20B007 AND CS20B008

1 Abstract

- In this project, we introduce a new, useful and interesting construct to extend the Java language. We propose an **advanced version of the switch case - switchAdv**. This version of the switch-case will help programmers to reduce the length and complexity of their code in many situations. It is also more efficient in terms of time complexity than its counterpart, the if - else if - else block.
- In many situations, we have to deal with a large number of conditions, like $x < 0, x < 2, x < 4, ...x < 10$ (x is a floating point number). In such situations, the normal switch-case cannot be used, since each case in it needs to have a particular value of x. We can overcome this by using the if-else if-else construct, but it would become quite lengthy. So we propose an advanced switch-case in which each case has a conditional expression attached to it, if that expression evaluates to be true, then we execute that particular case and exit from the switch-case block.
- We have also introduced priorities for the cases, with priorities decreasing from top to bottom. To explain the priority scheme, consider a situation where the condition associated with both case 2 and case 5 evaluate to be true, then, in this case, case 2 will be executed and then we will exit the switch-case block.
- This new construct will have a powerful impact in decreasing the length and complexity of the code, and a large number of conditionals would be dealt with in a compact piece of code. Also, the internal priority scheme will be helpful in many situations in which one case has precedence over another case but conditions associated with both cases evaluate to be true.
- Highly useful for writing programs for mathematical functions/graphs which behave differently in different regions(domains).

2 Motivation

2.1 Motivation-1

Consider the following situation:
Suppose n is an integer variable, and the following statements have decreasing priority:

- If $n > -2$, we have to execute the function f_1.
- If $n > 0$, we have to execute the function f_2.

- If $n > 2$, we have to execute the function f_3.
- If $n > 4$, we have to execute the function f_4.
- If neither is true, we have to execute the function f_5.

```
// implementation using if-else if-else
    construct
if(n > -2){
    //f1
    System.out.println("execute f1");
}
else if(n > 0){
    //f2();
    System.out.println("execute f2");
}
else if(n > 2){
    //f3();
    System.out.println("execute f3");
}
else if(n > 4){
    //f4();
    System.out.println("execute f4");
}
else {
    //f5();
    System.out.println("execute f5");
}
```

```
// implementation using switchAdv
switchAdv(){
 case n > -2 : //f1()
    System.out.println("execute f1");
 case n > 0  : //f2()
    System.out.println("execute f2");
 case n > 2  : //f3();
    System.out.println("execute f3");
 case n > 4  : //f4();
    System.out.println("execute f4");
 case default : //f5();
    System.out.println("execute f5");
}
```

As we can see the big if-else if-else code reduces to a small block. The number of lines reduced from 20 to 12 which corresponds to a 40 % decrease.

2.2 Motivation-2

The switchAdv will be very useful for mathematical programmers - for writing programs for graphs. In many cases, graphs behave differently in different ranges (see the example below). Using the switchAdv block makes the code compact and easily readable as compared to using the if-else if-else construct.

$$f(x) = \begin{cases} f_1(x), & \text{x} \leq 0 \\ f_2(x), & 2 \geq x \geq 0 \\ f_3(x), & 4 \geq x \geq 2 \\ f_4(x), & 8 \geq x \geq 4 \\ f_5(x), & 19 \geq x \geq 8 \\ f_6(x), & 20 \geq x \geq 19 \\ f_7(x), & 21 \geq x \geq 20 \\ f_8(x), & x \geq 21 \end{cases}$$

2.3 Motivation-3

Most decimal fractions cannot be represented exactly as binary fractions. A consequence of this is decimal floating-point numbers entered by the user are approximated by binary floating-point numbers and stored in the machine in this manner. Therefore it is risky to use floating point numbers in the normal switch-case. This issue does not arise in our switchAdv block. For example, to select the case associated with value x= 4/3, we simply make our condition as $1.33333333 < x < 1.33333334$ and we get the correct result.

2.4 Advantages and drawbacks

Advantages of switchAdv:

- More efficient in terms of time complexity than if-else if-else construct.
- Reduces length of code.
- Increases readability.
- Introduces the concept of priority, which can be used in many situations.
- Can be thought of as many normal switch-case cases packed into a single case.
- Very useful for situations for which in different continuous ranges of values, different operations are to be performed.
- Writing programs for graphs.

The one drawback of this switchAdv is that no more than one case can be executed, that is even if more than one conditional expression evaluates to be true, only the case associated with the first one will be considered.

3 Syntax

switchAdv(){
    case e_1 : e_2
    case e_3 : e_4
    case e_5 : e_6
    ....
    ....
    ....
    case e_{2n-1} : e_{2n}
    case default : e_{2n+1}
}

4 Type Checking

$$\forall i, i \geq 1 \text{ and } i \leq n$$
$$e_{2i-1} : Bool$$

5 Implementation Plan

For the implementation of our switchAdv construct, we implement two functions. The steps are as follows:

- The first function(getArrayFromswitchAdv) will take the syntax as input and check if the 'default' case is present. If the 'default' case is present, 'default' will be replaced with an expression: $e_0 = true$, if not present, no need to do anything.
- Now we will create an array A of the data type $pair < bool, expression >$.
  1. Default case was present:
     A = {(e_1, e_2), (e_3, e_4)....(e_{2n-1}, e_{2n}), (e_0, e_{2n+1})}
  2. Default case was not present:
     A = {(e_1, e_2), (e_3, e_4), (e_5, e_6)....(e_{2n-1}, e_{2n})}
- Now we will pass this array A to another function(evalRecursively) which will be a recursive function. It will do the following:
  1. Check if the array is empty, if yes, then return an empty expression(an expression that does nothing).
  2. Pick out the first element of the array. Say it is (e_i, e_j), if $e_i \Downarrow true$, return e_j.
  3. Remove/pop the first element of A.
  4. Call this same function with the argument as the updated array. Here is the pseudo-code of the above recursive function:

```
// implementation of evalRecursively
evalRecursively(A){
    if(A.empty())return E; // E is an empty expression
    firstElement = A.front();
    condition = firstElement.first();
    if(condition) return firstElement.second();
    A.pop_front();
    return evalRecursively(A);
}
```

- This function will return the required expression to the first function, which will finally return the required expression.

Another way to implement is by using a **Randomised Algorithm**:

- Suppose we have n cases, so n conditions: $C_1, C_2, \ldots . C_n$.
- Now we keep randomly choosing a condition from the set of conditions that have not been evaluated yet till we get a true condition, we also keep storing the final output of each condition in a hash table to avoid re-computation, once we get a condition that evaluates to true, we discard all other conditions below that condition. We can do this due to our priority scheme.

- We keep doing the above step until only a small finite number of conditions remain. We then evaluate the remaining conditions to get the final result.

This algorithm's worst-case time-complexity matches with the first algorithm but the average-case time complexity will be better than the first algorithm. The average-case time complexity of this algorithm will also be better than that of the equivalent if-else if-else block.

6 Semantics

We can think of switchAdv as a function which takes an array as input. Let's call this function E and the input array A. So the result of the switchAdv is E(A).

1. If A is an empty list, $E(A) = E_0$.(E_0 is the empty expression and can be thought of as a semi-colon).
2. If $A = \{(e_1,e_2), (e_3,e_4), (e_5,e_6)...(e_{2n-1},e_{2n})\}$, then A = $\{(e_1,e_2)\}$ + A', where $A' = \{(e_3,e_4), (e_5,e_6)...(e_{2n-1},e_{2n})\}$, if e_1 evaluates to be true , then we return e_2, else we return $E(A')$.

So formally,

$$\frac{A = \{\}}{E(A) \Downarrow E_0}$$

$$\frac{A = \{(e_1, e_2)\} + A' \quad e_1 \Downarrow true}{E(A) \Downarrow e_2}$$

$$\frac{A = \{(e_1, e_2)\} + A' \quad e_1 \Downarrow false}{E(A) \Downarrow E(A')}$$

These 3 rules describe the semantics of switchAdv.

7 Prior Work

Java 13 introduces an enhanced version of Switch which has many advantages over the traditional switch case. The traditional version suffers from drawbacks such as default fall-through due to missing break, only single values per case are supported. The enhanced version overcomes these issues and provides new features. Some of them are: supporting multiple values per case, allowing switch to return values (using yield keyword), improved scope rules and preview feature.