# ICSI 311 Assignment 9 – Interpreter Part 3

**This assignment is extremely important – (nearly) every assignment after this one uses this one!**

**If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.**

**You must submit .java files. Any other file type will be ignored. <u>Especially</u> ".class" files.**

**You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.**

**You must submit every file for every assignment.**

***<u>You must submit buildable .java files for credit.</u>***

## Introduction

We are going to keep working our way to more and more complex nodes in this assignment, but what you will find is that the previous work (like GetIDT) will make the more complex nodes simpler. In this assignment, we will deal with statements.

We run into a fundamentally difficult problem when we try to consider **break**, **continue** and **return** (in functions) because these change the flow of execution. You might ask why these are different from (say) a **while** or an **if**. The reason is because of how the interpreter works. Our blocks, functions and loops/conditionals have LinkedList<StatementNode>, so we have some piece of code:

```
InterpretListOfStatements  (/* some parameters */) {
      for (var s : statements) {
            processStatement(s);
      }
}
```

When you hit (for example) a while statement inside processStatement, it will look something like:

```
while (isTrue(evaluate(myWhile.Condition))
      InterpretListOfStatemnts (myWhile.statements);
```

This mutual recursion (InterpretListOfStatemnts calls processStatement which then calls InterpretListOfStatemnts) works well, so long as you never want to break out of the loop "unnaturally" (i.e. with a break or continue).

We will deal with this by making a new class which is the result of executing an instruction. It will hold both a string (the return value) and an enum – what happened, was it:

Normal, Break, Continue, Return

Quick review – break means that we terminate the loop, continue means we end this iteration of the loop early.

## Details

Create the new ReturnType class, making a constructor for only the enum and one for the enum and a string. Create an appropriate ToString().

Now create ProcessStatement:

ReturnType ProcessStatement(HashMap<String, InterpreterDataType> locals, StatementNode stmt)

Write code appropriate to the actual type of the statement:

**AssignmentNode**: Use GetIDT to evaluate the left and right side, set left's value equal to the GetIDT(right). Return type None, and the value of right

**BreakNode**: return with a return type of break

**ContinueNode**: return with a return type of continue

**DeleteNode**: get the array from the variables (local, then global). If indices is set, delete them from the array, otherwise delete them all.

**DoWhileNode**: call InterpretListOfStatements (we will write this in a little bit) in a do-while loop, using GetIDT to evaluate the condition. Check the return value of InterpretListOfStatements – if it is Break, then break out of the loop, on return, return from ProcessStatement.

**ForNode**: If there is an initial, call processStatement on it. Then create a while loop, using the forNode's condition as the while's condition. Inside, call InterpretListOfStatements() on forNode's statements. Same as DoWhile – check the return code and do the same thing. Make sure you call processStatement() on the forNode's increment.

ForEachNode: Find the array, loop over every key in the array's hashMap. Set the variable to the key, then call InterpretListOfStatements on the forEach's statements. Follow the same return rules as doWhile.

**FunctionCallNode**: much like last assignment, call RunFunctionCall().

**IfNode**: Remember that ifNodes are a linked list. Walk the linked list, looking for an IfNode where Condition is empty OR it evaluates to true. When you find that, call InterpretListOfStatements on ifNode.statements. If the return from InterpretListOfStatements is not "None" then return, passing that result back to the caller. Why? Consider this code:

```
while (!done) {
        if (a==5) break;
```

If a is 5, you want to break out of the while. To do that, the processing of the if node's statements must pass back the return type.

**ReturnNode**: if there is a value, evaluate it. Make a ReturnType of (value, Return).

**WhileNode**: much like doWhile, but with a while loop instead of a do-while.

Any other node type encountered should be an exception with a  good error message.

Now create IntepretListOfStatements:

ReturnType InterpretListOfStatements(LinkedLIst<StatementNode> statements, HashMap<String, InterpreterDataType> locals)

This is a simple loop over statements, calling processStatement() for each one, **except** that you should check the return type from each processStatement – if it is not None, return passing "up" the same ReturnType.

Testing on this assignment is very tedious. We will skip it and test more heavily next assignment when we can leverage the lexer and parser.

| Rubric | Poor | OK | Good | Great |
|---|---|---|---|---|
| Code Style | Few comments, bad names (0) | Some good naming, some necessary comments (3) | Mostly good naming, most necessary comments (6) | Good naming, non-trivial methods well commented, static only when necessary, private members (10) |
| ReturnType | None (5) | | | Members, constructor and ToString all correct (5) |
| ProcessStatement assignment | None (5) | | | Evaluates both sides, sets the new value, returns appropriately (5) |
| ProcessStatement break | None (5) | | | Returns appropriately (5) |
| ProcessStatement continue | None (5) | | | Returns appropriately (5) |
| ProcessStatement delete | None (5) | | | Deletes from array appropriately (10) |
| ProcessStatement doWhile | None (5) | | | Loops appropriately, handles break correctly (10) |
| ProcessStatement for | None (5) | | | Loops appropriately, handles break correctly (10) |
| ProcessStatement foreach | None (5) | | | Loops appropriately, handles break correctly (10) |
| ProcessStatement functionCall | None (5) | | | Calls function and passes along return value (5) |
| ProcessStatement if | None (5) | | | Loops appropriately, passes along return value correctly (10) |

|  |  |  |  |  |
|---|---|---|---|---|
| ProcessStatement return | None (5) |  |  | Returns appropriately (5) |
| ProcessStatement while | None (5) |  |  | Loops appropriately, handles break correctly (10) |