COMP0012 - Coursework Part II - Report

1. Optimisation Algorithm

Each of the implementations below must be implemented and called from the ConstantFolder class in the *optimize()* function body. To enable collaboration and reduce merge conflicts, the functions corresponding to their implementation task are called in this *optimize()* function:

- *simpleVariableFoldingMethod(ClassGen cgen, ConstantPoolGen cpgen);*
- *constantVariableFoldingMethod(ClassGen cgen, ConstantPoolGen cpgen);*
- *dynamicVariableFoldingMethod(ClassGen cgen, ConstantPoolGen cpgen);*

If implemented separately, each method would have a different implementation of the *optimize()* method, hence, each method has a corresponding method called from *optimize()* that allows for this to be abstracted out.

1.1 Simple Folding

The simple folding algorithm aims to realize constant folding optimization for types including int, long, float and double. By identifying and performing constant evaluations during compiling, the algorithm is able to reduce time needed at run time.

The simple folding algorithm is called in the *optimize()* method of the ConstantFolding.java file, with the name *simpleVariableFoldingMethod*. First, it iterates over all methods defined in the targeted class, and creates an *InstructionList* object to store all the bytecode instructions for the current method. Then the simple folding methods for different data types are called (*simpleInt* for type int, *simpleLong* for type long, *simpleFloat* for type float, and *simpleDouble* for type Double).

For each of these methods (which shares the same logic), an *InstructionFinder* object is created to find a specific regular expression where two constants (like *ICONST*) are followed by an arithmetic operation (*IADD*, *ISUB*, *IMUL* or *IDIV*). The finder finds all matching patterns and extracts the constants using *getConstant()*. After that, the corresponding arithmetic operation is performed, and the result is calculated. Finally, the correct replacing instruction is selected, and *replaceInst()* is used to conduct the replacement.

The generic method *getConstant()* is able to handle different constant types included above. Depending on the specific type it received, it either loads the required constant from the constant pool, or gets the constant value directly. On the other hand, the *replaceInst()* method iterates over the instructions that need to be replaced. For each instruction, *setInstruction()* is called to replace the old instruction with the new one.

1.2 Constant Variables

Constant variable folding optimizes local variables (int, long, float, double) which are only assigned to once (i.e. not modified) within a method. These variables are eligible for constant propagation across the method body which reduces the number of JVM bytecode instructions.

The algorithm for this optimization is primarily implemented in `simulateInstructionList` which is called from `constantVariableFoldingMethod` inside `ConstantFolder.java`. The algorithm works by iterating over each method in the class and optimizing within each method. It works by simulating the execution of relevant bytecode instructions.

The algorithm (inside of `simulateInstructionList`) starts by iterating over each instruction to find and record all assignments by tracking the local variable index and its value. Assignments are denoted in the bytecode as a stack push type instruction (i.e. BIPUSH, LCONST, LDC_W, etc.) immediately followed by a store type instruction (i.e. ISTORE, LSTORE, etc.). From this we can determine which variables are not reassigned and hence we can mark these for optimization in the next step. We then propagate these (non reassigned) variables by replacing load type instructions with push type ones where possible.

Finally, our updated instruction list is committed via `cgen.replaceMethod`. Method attributes such as `setMaxStack` and `setMaxLocals` are also updated to reflect changes in the stack and local variables.

In practice, this algorithm is implemented by declaring a stack to simulate the JVM operand stack, and a hashmap named `locals`, which tracks the local variable array. We iterate over each instruction and then call methods to handle each instruction type which also mutates the state of the stack and locals. These methods are `handleLoad`, `handleStore`, `handlePush`, `handleCasts`, `handleLdc`, `handleOperation` and `handleComparisons`.

These *handle* methods take the current instruction and returns true if that instruction was processed and false if it was unrecognized. If all the methods fail to recognize the instruction then we cannot optimize this line via constant folding and trying to do so would likely result in an error hence we skip this instruction using a continue statement.

Finally, if the instruction is a return type instruction and the value we return has been folded into a constant with no side effects then we can optimize this by replacing the whole instruction list with push type followed by a return type instruction. Otherwise we return the optimized instruction list as is.

1.3 Dynamic Variables

The Dynamic Variable Folding algorithm aims to optimise local variables of type int, float, long and double, where the variables will be reassigned to different constant values. To perform this, constant values of the variables are tracked, and variable loads are substituted with direct constant loads.

Firstly, *cgen.getMethods()* is used to iterate through every method in the class. For each method, a hash map is initialised to track the constant values of the integers, where the keys are variable indices and the values are constant values. Also, the method's instruction list is being accessed by *mg.getInstructionList()*, which, if it has no instructions, this method is skipped.

Next, each instruction in the instruction list is scanned. If the instruction is a constant loading instruction (*ICONST*, *BIPUSH*, or *SIPUSH*) followed by a store instruction (*ISTORE*), the constant value is recorded in the hash map with the variable's index.

If the instruction is a variable loading instruction (*ILOAD*), it checks if the variable is in the hash map or not. If so, the loading instruction is being replaced with an appropriate constant-pushing instruction, which is based on the value. *ICONST* for values between -1 and 5, *BIPUSH* for values between -128 and 127, *SIPUSH* for values between -32768 and 32767 and *LDC* for any other values. The modified flag is then set to true to indicate the replacement.

If the instruction is a variable storing instruction (*ISTORE*), the variable's index is removed from the hashmap, as its value may no longer be a constant.

Lastly, if any replacement, which is indicated by the modified flag, was made earlier, the method's stack size, instruction positions, local variable count have to be updated as they are different after optimization. After all, the original method is then replaced by the optimized version with *cgen.replaceMethod().*

2. Testing Platform

All software must be tested on a variety of release platforms to ensure that no platform-specific bugs can occur. To honour this practice, we tested both the unit tests and bytecode outputs on the following systems:

- Linux (through WSL)
- MacOS

Unfortunately, due to *ant* requiring a Linux-based file system, and the lab machines not having *ant* installed, we could not test with native Windows or Linux.

2.1 MacOS

Here is the MacOS platform tests, running the *ant* command and ensuring our optimisation resulted in a reduction of the number of bytecode instructions.

2.1.1 ant Build and Unit Tests

On MacOS we can see that all unit tests pass, with the build being successful as defined in the handout. See below for the output of building the project and running the unit tests by issuing the command as defined by the handout.

```
~/Dev/year3/o/comp0012/cw2 main ❯ ant                                          11:31:06
Buildfile: /Users/Benjamin/Dev/year3/open/comp0012-compilers/cw2/build.xml

compile.source:
    [javac] Compiling 1 source file to /Users/Benjamin/Dev/year3/open/comp0012-compilers/cw2/build/classes

generate:
     [java] Generated: /Users/Benjamin/Dev/year3/open/comp0012-compilers/cw2/build/classes/comp0012/target/SimpleFolding.class

optimise:
     [echo] Running constant folding optimisation...
     [java] Running COMP207p courswork-2

compile.tests:

compile:

test.original:
     [echo] Running unit tests for the original classes...
    [junit] Running comp0012.target.ConstantVariableFoldingTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 sec
    [junit] Running comp0012.target.DynamicVariableFoldingTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 sec
    [junit] Running comp0012.target.SimpleFoldingTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec

test.optimised:
     [echo] Running unit tests for the optimised classes...
    [junit] Running comp0012.target.ConstantVariableFoldingTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec
    [junit] Running comp0012.target.DynamicVariableFoldingTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 sec
    [junit] Running comp0012.target.SimpleFoldingTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.005 sec

test:

BUILD SUCCESSFUL
Total time: 1 second
```

2.1.2 SimpleFolding Bytecode Optimisation

Shown below are the optimisations that our code produced from the original bytecode classes, showing a reduction in the number of instructions. We aimed for each function to

simply have 2-3 instructions being a load, a constant, and a return, however, this was not fully realised as shown below in public void simple();

```
~/Dev/year3/open/comp0012-compilers/cw2 main !5 ❯ javap -c optimised.classes.comp0012.target.SimpleFolding
Warning: File ./optimised/classes/comp0012/target/SimpleFolding.class does not contain class optimised.classes.comp0012.target.SimpleFolding
Compiled from "Type1.j"
public class comp0012.target.SimpleFolding {
  public comp0012.target.SimpleFolding();
    Code:
       0: aload_0
       1: invokespecial #17                 // Method java/lang/Object."<init>":()V
       4: return

  public void simple();
    Code:
       0: getstatic     #12                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #24                 // int 67
       5: ldc           #8                  // int 12345
       7: iadd
       8: invokevirtual #7                  // Method java/io/PrintStream.println:(I)V
      11: return
}
```

2.1.3 ConstantVariableFolding Bytecode Optimisation

Constant variable folding was implemented completely, with each optimised bytecode output containing the minimum required instructions to return a constant value. See below for the bytecode generated by our optimisation

```
~/Dev/year3/open/comp0012-compilers/cw2 main !5 ❯ javap -c optimised.classes.comp0012.target.ConstantVariableFolding
Warning: File ./optimised/classes/comp0012/target/ConstantVariableFolding.class does not contain class optimised.classes.comp0012.target.ConstantVariableFolding
public class comp0012.target.ConstantVariableFolding {
  public comp0012.target.ConstantVariableFolding();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public int methodOne();
    Code:
       0: ldc           #25                 // int 3650
       2: ireturn

  public double methodTwo();
    Code:
       0: ldc2_w        #26                 // double 1.67d
       3: dreturn

  public boolean methodThree();
    Code:
       0: iconst_0
       1: ireturn

  public boolean methodFour();
    Code:
       0: iconst_1
       1: ireturn
}
```

2.1.4 DynamicVariableFolding Bytecode Optimisation

Dynamic variable folding was partially successful, with 3 / 5 of the test functions being reduced down to the minimum required instructions. The 2 remaining functions could not be optimised due to an incorrect implementation of goto and boolean handling. See below for the bytecode generated by our optimisation

```
~/Dev/year3/open/comp0012-compilers/cw2 main !5 ❯ javap -c optimised.classes.comp0012.target.DynamicVariableFolding
Warning: File ./optimised/classes/comp0012/target/DynamicVariableFolding.class does not contain class optimised.classes.comp0012.target.DynamicVariableFolding
public class comp0012.target.DynamicVariableFolding {
  public comp0012.target.DynamicVariableFolding();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public int methodOne();
    Code:
       0: ldc           #40                 // int 1301
       2: ireturn

  public boolean methodTwo();
    Code:
       0: sipush        12345
       3: istore_1
       4: ldc           #7                  // int 54321
       6: istore_2
       7: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      10: iload_1
      11: iload_2
      12: if_icmpge     19
      15: iconst_1
      16: goto          20
      19: iconst_0
      20: invokevirtual #14                 // Method java/io/PrintStream.println:(Z)V
      23: iconst_0
      24: istore_2
      25: iload_1
      26: iload_2
      27: if_icmple     34
      30: iconst_1
      31: goto          35
      34: iconst_0
      35: ireturn

  public int methodThree();
    Code:
       0: ldc           #41                 // int 84
       2: ireturn

  public int methodFour();
    Code:
       0: ldc           #20                 // int 534245
       2: istore_1
       3: iload_1
       4: sipush        1234
       7: isub
       8: istore_2
       9: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      12: ldc           #21                 // int 120298345
      14: iload_1
      15: isub
      16: i2d
      17: ldc2_w        #22                 // double 38.435792873d
      20: dmul
      21: invokevirtual #24                 // Method java/io/PrintStream.println:(D)V
      24: iconst_0
      25: istore_3
      26: iload_3
      27: bipush        10
      29: if_icmpge     49
      32: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      35: iload_2
      36: iload_1
      37: isub
      38: iload_3
      39: imul
      40: invokevirtual #27                 // Method java/io/PrintStream.println:(I)V
      43: iinc          3, 1
      46: goto          26
      49: iconst_4
      50: istore_1
      51: iload_1
      52: iconst_2
      53: iadd
      54: istore_2
      55: iload_1
      56: iload_2
      57: imul
      58: ireturn
}
```

2.2 Windows (WSL)

We also ran all of the previously demonstrated commands on Windows WSL to ensure that a Linux-based platform also performed as expected. As we can see by the following screenshots, the behaviour of the program did not change based on the platform it was ran on, ensuring that our program is multi-platform capable.

2.2.1 ant Build and Unit Tests

Here we can see that on WSL, our optimisation program passes all unit tests and builds successfully

```
(base) tim@tym:/mnt/e/dev/uni/COMP0012-CW2$ ant
Buildfile: /mnt/e/dev/uni/COMP0012-CW2/build.xml

compile.source:

generate:
WARNING: A terminally deprecated method in java.lang.System has been called
WARNING: System::setSecurityManager has been called by org.apache.tools.ant.types.Permissions (file:/usr/share/ant/lib/ant.jar)
WARNING: Please consider reporting this to the maintainers of org.apache.tools.ant.types.Permissions
WARNING: System::setSecurityManager will be removed in a future release
     [java] Generated: /mnt/e/dev/uni/COMP0012-CW2/build/classes/comp0012/target/SimpleFolding.class

optimise:
     [echo] Running constant folding optimisation...
     [java] Running COMP207p courswork-2
     [java] Unsupported instruction: getstatic[178](3) 8
     [java] Cannot optimize this instruction; skipping constant folding.
     [java] Unsupported instruction: getstatic[178](3) 8
     [java] Cannot optimize this instruction; skipping constant folding.
     [java] Unsupported instruction: getstatic[178](3) 12
     [java] Cannot optimize this instruction; skipping constant folding.

compile.tests:

compile:

test.original:
     [echo] Running unit tests for the original classes...
    [junit] Running comp0012.target.ConstantVariableFoldingTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.049 sec
    [junit] Running comp0012.target.DynamicVariableFoldingTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.054 sec
    [junit] Running comp0012.target.SimpleFoldingTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.048 sec

test.optimised:
     [echo] Running unit tests for the optimised classes...
    [junit] Running comp0012.target.ConstantVariableFoldingTest
    [junit] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.035 sec
    [junit] Running comp0012.target.DynamicVariableFoldingTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.05 sec
    [junit] Running comp0012.target.SimpleFoldingTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.042 sec

test:

BUILD SUCCESSFUL
Total time: 4 seconds
```

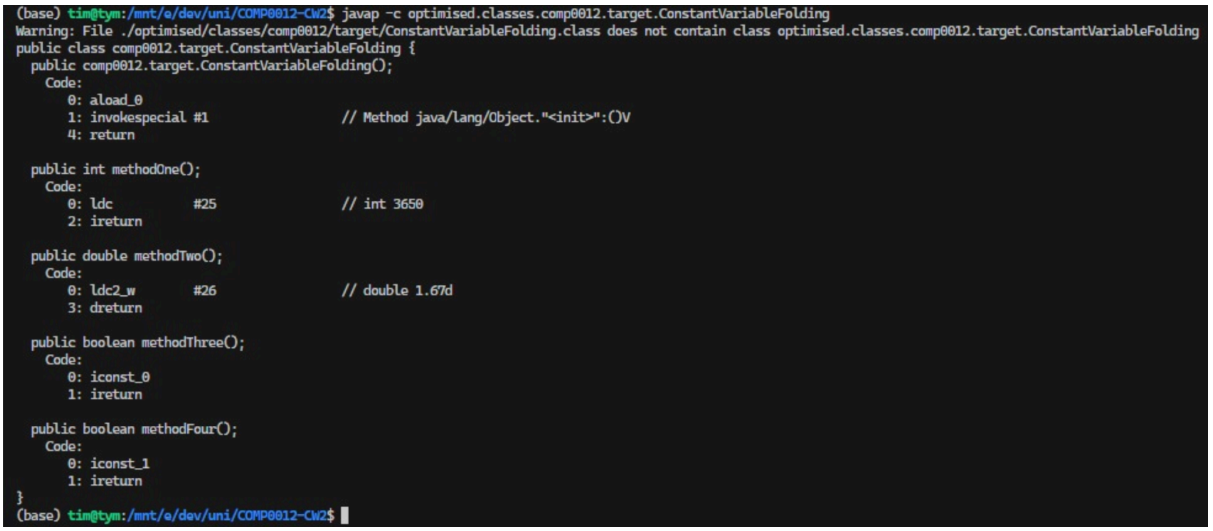
2.2.2 SimpleFolding Bytecode Optimisation

As previously shown, running the simple folding optimisation reduces the number of bytecode instructions resulting in the same output as MacOS

```
(base) tim@tym:/mnt/e/dev/uni/COMP0012-CW2$ javap -c optimised.classes.comp0012.target.SimpleFolding
Warning: File ./optimised/classes/comp0012/target/SimpleFolding.class does not contain class optimised.classes.comp0012.target.SimpleFolding
Compiled from "Type1.j"
public class comp0012.target.SimpleFolding {
  public comp0012.target.SimpleFolding();
    Code:
       0: aload_0
       1: invokespecial #17                 // Method java/lang/Object."<init>":()V
       4: return

  public void simple();
    Code:
       0: getstatic     #12                 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #24                 // int 67
       5: ldc           #8                  // int 12345
       7: iadd
       8: invokevirtual #7                  // Method java/io/PrintStream.println:(I)V
      11: return
}
```

2.2.3 ConstantVariableFolding Bytecode Optimisation

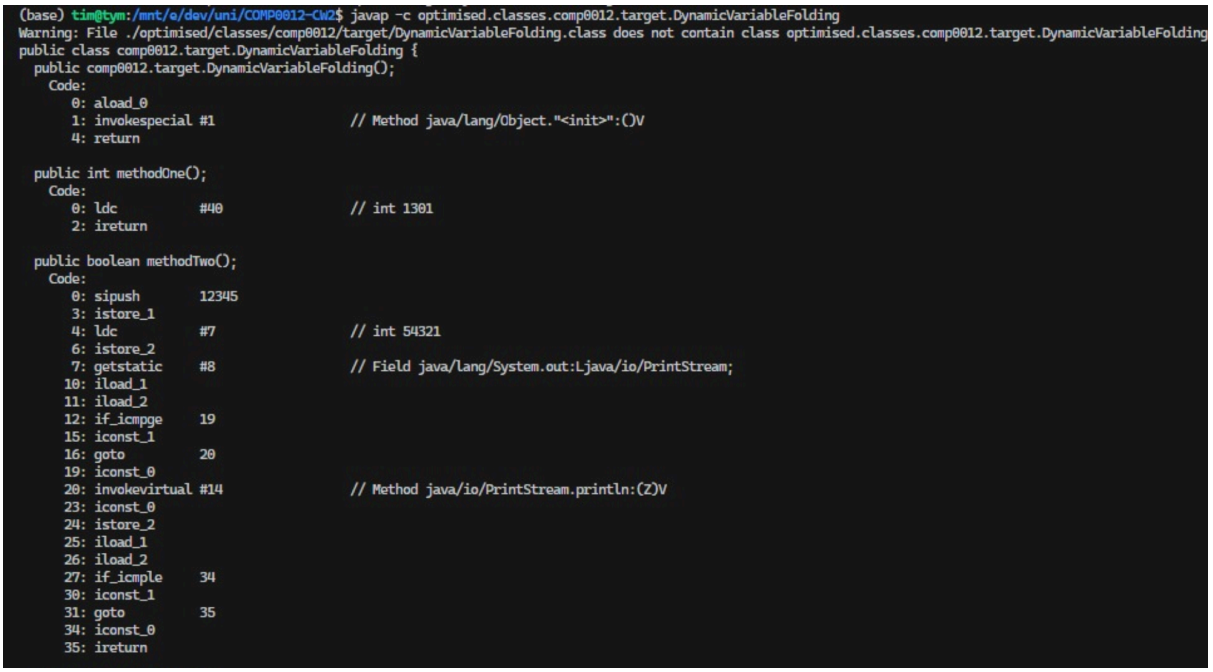
Again, we have the same output as shown on MacOS



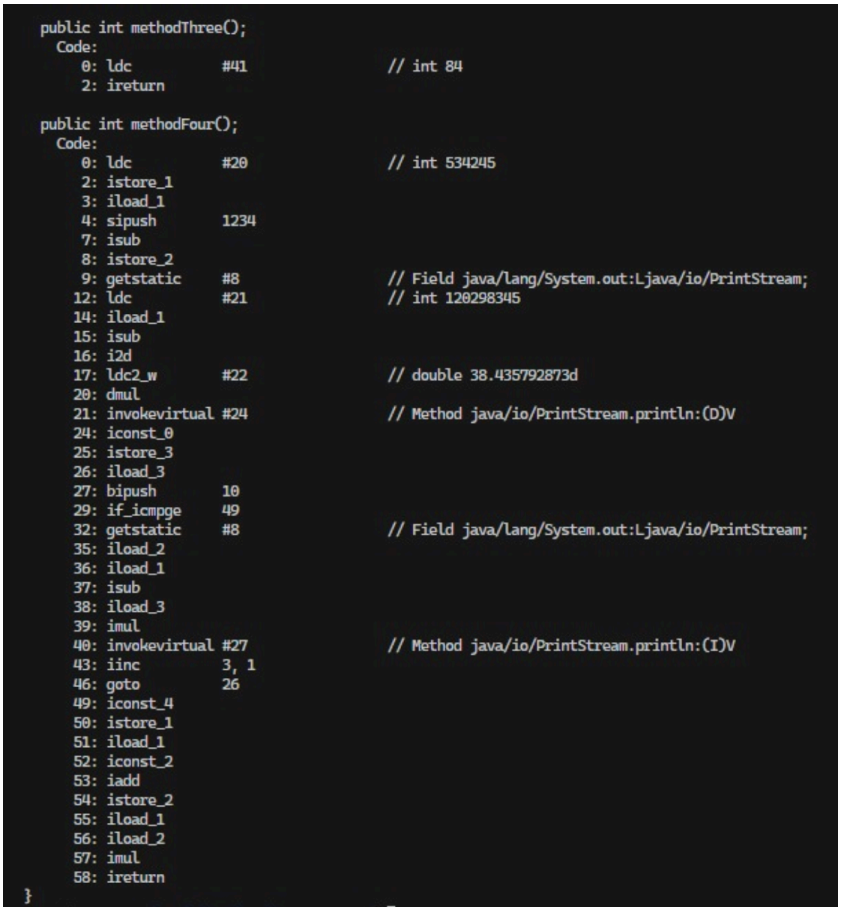
```
(base) tim@tym:/mnt/e/dev/uni/COMP0012-CW2$ javap -c optimised.classes.comp0012.target.ConstantVariableFolding
Warning: File ./optimised/classes/comp0012/target/ConstantVariableFolding.class does not contain class optimised.classes.comp0012.target.ConstantVariableFolding
public class comp0012.target.ConstantVariableFolding {
  public comp0012.target.ConstantVariableFolding();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public int methodOne();
    Code:
       0: ldc           #25                 // int 3650
       2: ireturn

  public double methodTwo();
    Code:
       0: ldc2_w        #26                 // double 1.67d
       3: dreturn

  public boolean methodThree();
    Code:
       0: iconst_0
       1: ireturn

  public boolean methodFour();
    Code:
       0: iconst_1
       1: ireturn
}
(base) tim@tym:/mnt/e/dev/uni/COMP0012-CW2$
```

2.2.4 DynamicVariableFolding Bytecode Optimisation

Finally, regarding dynamic variable folding, we get the same output that we have previously seen on MacOS



```
(base) tim@tym:/mnt/e/dev/uni/COMP0012-CW2$ javap -c optimised.classes.comp0012.target.DynamicVariableFolding
Warning: File ./optimised/classes/comp0012/target/DynamicVariableFolding.class does not contain class optimised.classes.comp0012.target.DynamicVariableFolding
public class comp0012.target.DynamicVariableFolding {
  public comp0012.target.DynamicVariableFolding();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public int methodOne();
    Code:
       0: ldc           #40                 // int 1301
       2: ireturn

  public boolean methodTwo();
    Code:
       0: sipush        12345
       3: istore_1
       4: ldc           #7                  // int 54321
       6: istore_2
       7: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      10: iload_1
      11: iload_2
      12: if_icmpge     19
      15: iconst_1
      16: goto          20
      19: iconst_0
      20: invokevirtual #14                 // Method java/io/PrintStream.println:(Z)V
      23: iconst_0
      24: istore_2
      25: iload_1
      26: iload_2
      27: if_icmple     34
      30: iconst_1
      31: goto          35
      34: iconst_0
      35: ireturn
```



```
  public int methodThree();
    Code:
       0: ldc           #41                 // int 84
       2: ireturn

  public int methodFour();
    Code:
       0: ldc           #20                 // int 5342145
       2: istore_1
       3: iload_1
       4: sipush        1234
       7: isub
       8: istore_2
       9: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      12: ldc           #21                 // int 120298345
      14: iload_1
      15: isub
      16: i2d
      17: ldc2_w        #22                 // double 38.435792873d
      20: dmul
      21: invokevirtual #24                 // Method java/io/PrintStream.println:(D)V
      24: iconst_0
      25: istore_3
      26: iload_3
      27: bipush        10
      29: if_icmpge     49
      32: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
      35: iload_2
      36: iload_1
      37: isub
      38: iload_3
      39: imul
      40: invokevirtual #27                 // Method java/io/PrintStream.println:(I)V
      43: iinc          3, 1
      46: goto          26
      49: iconst_4
      50: istore_1
      51: iload_1
      52: iconst_2
      53: iadd
      54: istore_2
      55: iload_1
      56: iload_2
      57: imul
      58: ireturn
}
```

3. Contribution

The project consists of four main tasks, each contributing to the overall process. The team has four members, with the responsibilities allocated as follows:

1. **Lucy:**
    - Was responsible for implementing **Simple Folding** (Task 1).
    - Crafted the report structure and wrote the contribution section of the report.

2. **Brian:**
    - Responsible for implementing **Constant Variables** (Task 2).

3. **Ben:**
    - Worked collaboratively on **Dynamic Variables** (Task 3) with Ivan
    - Integrated the code changes from all team members to ensure the overall project is cohesive, handling any merge conflicts.
    - Wrote the testing section of the report.

4. **Ivan:**
    - Worked collaboratively on **Dynamic Variables** (Task 3) with Ben.

Additionally, each team member is responsible for writing the **report related to their specific task**. In summary, the team works collaboratively with continuous communication to ensure the successful development of the final solution.