# COMP0012 - Compilers (MSA)
# Coursework Part II: Code Optimisation
Due on 16/04/2025, 16:00 UTC

**Guidelines** This coursework is compulsory. It will be graded on a scale of 0 to 10 and contribute 10% of the overall marks for the module. You should submit it online through Moodle by 16/04/2025, 16:00 UTC. Submissions received after this deadline will be considered late and the UCL late submission penalties will be applied (i.e. to reduce the mark awarded) [1].

## 1 Aim

Using Java and BCEL (Byte Code Engineering Library) [2] implements *peephole optimisation* as much as possible. The Java compiler already does some of this. For example, the following Java code

```
. . .
int a = 429879283 - 876987;
// no assignment to a in the middle
System.out.println(527309 - 1293 + 5 * a );
. . .
```

produces the following constant pool in the corresponding Java bytecode .class file:

```
. . .
const #2 = int 429002296; // this would be variable a
. . .
const #7 = int 526016; // this would be the subexpression inside println
. . .
```

We can see that `javac` has performed two arithmetic operations to achieve constant folding: $429879283-876987 = 429002296$, and $527309 - 1293 = 526016$. However, if there is no assignment to the variable $a$, it can be folded further by propagating the value of $a$. Your peephole optimisation should identify such patterns and perform constant folding as much as possible.

## 2 Tasks and Evaluation

This coursework will be number-graded (0 to 10) and contribute to 10% of the overall course outcome. There are four sub-goals for the optimisation. Each sub-goal will result in specific patterns of bytecode after compilation. Your task is to implement a single bytecode optimiser that identifies and optimises all four of them. Marks will be awarded based on the achievement of each sub-goal:

- **Simple Folding (2 marks)**: The first sub-goal is to perform constant folding for the values of type int, long, float, and double, in the bytecode constant pool. Note that the Java compiler usually performs this for you. You will be provided with an artificial class file, which contains an un-optimized constant pool.
- **Constant Variables (3 marks)**: The second sub-goal requires you to optimize uses of local variables of type int, long, float, and double, whose value does not change throughout the scope of the method (i.e. after the declaration, the variable is not re- assigned). You need to propagate the initial value throughout the method to achieve constant folding. For example, you will be targeting something like the following:

```
public int optimiseMe (){
  int a = 534245;
  int b = a - 1234;
  // the following argument can be folded into a constant
  System.out.println((120298345 - a ) * 38.435792873);
  for (int i = 0 ; i < 10 ; i++){
    // the subexpression (b - a) can be folded into a constant
    System.out.println((b - a) * i);
  }
  // the return value can be folded into a constant
  return a * b ;
}
```

- **Dynamic Variables (3 marks)**: This sub-goal requires you to optimise the use of local variables of type int, long, float, and double, whose values will be reassigned with a different constant numbers during the scope of the method. You still need to propagate the value of the variable, but for specific intervals: starting from the assignment (or initialisation) until the next assignment.
- **Report (2 marks)**: The final sub-goal requires you to present the approach, algorithm, implementation, and testing the above three sub-goals. Your report should be clear so that The reader is able to reproduce your implementation and testing results.

## 3 Project Structure

The coursework files provide a skeleton project with a *ant* build script and relevant Java libraries. Download comp0012-coursework2.zip from Moodle. Uncompressed, it contains the Java source code directory **src**, unit test directory **test**, dependency libraries **lib**, and configuration **build.xml**.

The project is organised as follows.

```
comp0012-coursework2
├── src .................................. Java source for optimisation and test targets
│   └── comp0012
│       ├── main
│       │   ├── Main.java ........................ Main recursively processes .class files
│       │   └── ConstantFolder.java .................... Optimises individual .class files
│       └── target
│           ├── SimpleFolding.j .. Jasmin code that generates artificial target for sub-goal 1
│           ├── ConstantVariableFolding.java ..................... Target for sub-goal 2
│           └── DynamicVariableFolding.java ...................... Target for sub-goal 3
├── test ................................................. JUnit test case sources
├── lib ........................................................... Library jars
└── build.xml ................................................ ant build script
```

The **src** directory contains

- **Main.java**: It contains the main entry of the optimiser. This function takes two arguments when compiled:

  ```
  java comp0012.main.Main -in [input-folder] -out [output-folder]
  ```

  Where `input-folder` contains the original classfiles and `output-folder` stores the optimised classfiles. In particular, `Main.java` includes code that

  - recursively visits all class files in all subfolders under the input folder,

  - invokes methods in **ConstantFoler.java** to optimize ***.class** files, and
  - writes the optimized classfiles in the corresponding directory structure under the output folder.

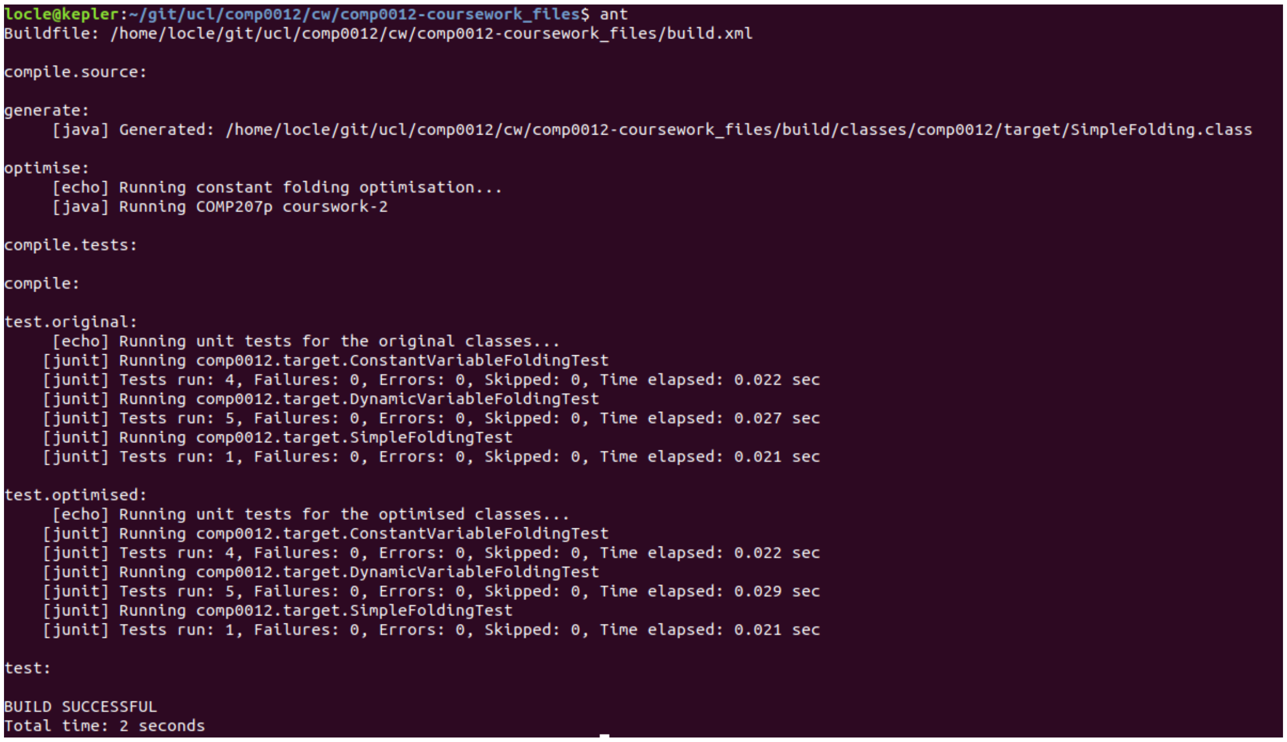
- **ConstantFoler.java**: It contains the methods that implement the optimisation sub-goals. Currently, it does not do any optimisation: it simply writes unoptimised, unmodified class files. Your task is to provide an implementation to achieve the four sub-goals above.
- The **target** folder: It includes three target files, each of which corresponds to one testing sample of the sub-goals.

The **test** folder contains unit tests used for both original targets and optimised targets. The **lib** directory contains all necessary dependencies to compile and execute the optimiser. Finally, **build.xml** contains the **ant** build script and test harnesses for both test.original and test.optimised tasks. For the former, it runs the unit tests using the original class files; for the latter, however, it sets the classpath to the optimised, thereby testing the optimised classes.

To compile the project, execute the optimiser, and run unit tests, simply run the following command in the root directory:

```
ant
```

If the build completes successfully, the output will be as the following sample.



```
locle@kepler:~/git/ucl/comp0012/cw/comp0012-coursework_files$ ant
Buildfile: /home/locle/git/ucl/comp0012/cw/comp0012-coursework_files/build.xml

compile.source:

generate:
     [java] Generated: /home/locle/git/ucl/comp0012/cw/comp0012-coursework_files/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: 0.022 sec
    [junit] Running comp0012.target.DynamicVariableFoldingTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 sec
    [junit] Running comp0012.target.SimpleFoldingTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 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: 0.022 sec
    [junit] Running comp0012.target.DynamicVariableFoldingTest
    [junit] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.029 sec
    [junit] Running comp0012.target.SimpleFoldingTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 sec

test:

BUILD SUCCESSFUL
Total time: 2 seconds
```

## 4 Deliverables

Each group should submit the following deliverables by the submission deadline:

- **Implementation**: A Java implementation of the optimisation. Use the given directory structure and the build script in the skeleton files (available from Moodle).
- **Report**: A written joint report that contains:
  - Briefly present your optimisation algorithm. Describe the optimisation you have implemented in as much detail as possible (there is no page limit). For each task, state clearly which part of the Java implementation (e.g., class names, method names) has implemented it; penalised 20% of the task if this information is not provided.

  - Platforms/OS your project was tested. Testing the submission on the department's Linux machines, Ubuntu 20.04 or Ubuntu 22.04, is strongly recommended. This is a good way to catch any platform-dependent fault in your script and configurations. Penalised 20% if this information is not provided.
  - Also, describe briefly each team member's contribution to the coursework. Members who contribute significantly less than other team members may have a penalty. If someone does not contribute at all, will get 0 marks. Penalised 10% if this information is not provided and it will be understood that each team member's contribution is equal.

You should implement the constant folding optimisation in the given **ConstantFolder.java**, while preserving the directory structure. The build script is designed so that simply issuing ant will compile the source code and unit test, generate **SimpleFolding.class**, execute the optimisation, and unit test both the original and the optimised classes. Your deliverable should support this process out of the box, and should follow the directory structure below:

```
groupXX.................................Name the directory as with your group name
 |_ report.pdf ........................... Your report, either in PDF or Word format
 |_ src ................................. Java source for optimisation and test targets
 |   |_ comp0012
 |       |_ main
 |       |   |_ Main.java.........................Main recursively processes .class files
 |       |   |_ ConstantFolder.java...................With your implementation added
 |       |_ target
 |           |_ SimpleFolding.j .. Jasmin code that generates artificial target for sub-goal 1
 |           |_ ConstantVariableFolding.java ..................... Target for sub-goal 2
 |           |_ DynamicVariableFolding.java ...................... Target for sub-goal 3
 |_ test.......................................................JUnit test case sources
 |_ lib .............................................................. Library jars
 |_ build.xml.................................................... ant build script
```

Compress the top-level directory (which should be named as **groupXX** where XX is your group number) and submit it through Moodle. The report should be included in the top-level directory. Make sure that you follow this directory structure instructions. **Violations will result in a reduction of points.**

Note that make sure your submission is self-contained. It should not depend on any file outside the submitted directory, such as files on your hard drive or online. We expect **ant** under **groupXX** directory simply to work, straight out of the box.

## 5 Tools

These are the relevant tools. `BCEL` is part of the coursework requirement (i.e. you are requested to use this to modify the bytecode); you will probably find the others helpful during the development.

- **Bytecode Engineering Library**: To make changes to the bytecode, it is better to use a library (direct, byte-level access is possible but would be very painful). Apache BCEL (Byte Code Engineering Library [2]) is a widely used tool. It comes with a detailed tutorial: `http://commons.apache.org/proper/commons-bcel/manual/manual.html`.
- **Bytecode Disassembler**: Java comes with a disassembler, javap, which takes a .class file and prints out the corresponding byte code representation. Try `javap -c -verbose YOURCLASS` to see the contents of `YOURCLASS.class`. This will allow you to observe bytecode patterns created by various program structures in Java.

- **Jasmin**: Jasmin(http://jasmin.sourceforge.net) is a Java assembler - you can write bytecode source codes in a way similar to writing MIPS assembler language, and Jasmin can compile it into Java classes. If you want to create fine-tuned bytecode instructions to test your optimisation, this is the right tool (combined with bytecode disassembler). Read `SimpleFolding.j` to get started, along with the online documentation.
- **Java Bytecode Reference**: for the full list of bytecode instructions in Java, see https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions.

## References

[1] UCL. Penalties for late submission of coursework. https://www.ucl.ac.uk/academic-manual/chapters/chapter-4-assessment-framework-taught-programmes/section-3-module-assessment#3.12. Online; accessed 20 January 2024.

[2] Apache. Apache Commons BCEL. http://commons.apache.org/proper/commons-bcel/. Online; accessed 20 January 2024.