Advanced Java Programming **2024**

Instructors: Dr. Andrea Rosà and Prof. Walter Binder TA: Shifat Sahariar Bhuiyan

---

DiSL Project (40 points + 10 bonus points)

In this project, you are required to implement multiple profilers using DiSL. Please read carefully the instructions (including the submission instructions) and the exercises below. If something is unclear, please post a message on the forum on iCorsi as soon as possible.

Instructions

This project is composed of 7 exercises (plus 1 bonus exercise). Each comprises an observed application (provided already compiled, without the Java source code) and a profiler (that you have to implement). The observed application and the profiler of one exercise are independent from the ones of other exercises.

On iCorsi, you can download a template for this project. The profiler required for solving exercise *i* must be implemented by adding its source code in src-profiler/ex*i* (note that this means that profiler classes should be contained in package ex*i*). The profiler can be composed of how many classes you like. However, all DiSL snippets must be added to the class `Instrumentation.java` (already provided in the template).

The JDK of reference for testing the profilers is `OpenJDK21`. Please do not use other JDKs.

Unless otherwise noted, **all profilers must be thread-safe**. In this case, *non-thread-safe implementations will receive **0** points.* Points are awarded proportionally to the correctness and efficiency of the solution.

Please refer to the instructions in the `README.md` file (in the template) for additional information about compiling, building and running the profiler for a given exercise and how to run the observed application.

Submission Instructions

You must implement the profilers by modifying/adding classes to the provided template as needed. Your submission must consist of an archive containing the modified project template. You are not required to submit any other document. You can specify your notes as comments in the profiler's source code if needed. There is no naming requirement for the submission.

The project must be submitted before the deadline published on iCorsi to be evaluated. Moreover, all code must compile without errors.

Note on Grading

Since the project is part of the final exam, the grade will not be communicated to you on iCorsi. After the final exam, you will receive the final course grade from the Dean’s Office.

Exercise 1 (5 points)

Implement a profiler that detects all accesses to instance fields, as well as the number of unique static fields accessed. The term “access” refers to either a read or a write from/to the field. The profiler must maintain three separate counters:

- Reads from instance fields
- Writes to instance fields
- Unique static fields accessed

At the end of the application execution, the profiler should print the following information:

```
Reads from instance fields: <a>
Writes to instance fields: <b>
Unique static field accessed: <c>
```

Replace `<a>` with the number of reads from instance fields, `<b>` with the number of writes to instance fields, and `<c>` with the number of unique static fields accessed.

Example Output:

```
Instance field read accesses: 446152
Instance field write accesses: 94654
Unique static fields accessed: 265
```

Note: Due to non-determinism, a correct implementation may print numbers significantly different to the ones above.

Exercise 2 (5 points)

Implement a profiler that detects all method invocations of the following types: **`invokestatic`**, **`invokespecial`**, **`invokevirtual`**, and **`invokedynamic`**, performed by threads of type `ex2.MyThread`. Upon the termination of a thread of this type, the profiler should print the following line:

```
Thread: <a> - #static: <b> - #special: <c> - #virtual: <d> - #dynamic: <e>
```

Replace `<a>` with the name of the thread, `<b>` with the number of **`invokestatic`** executed by the `<a>` thread, `<c>` with the number of **`invokespecial`** executed by the `<a>` thread, `<d>` with the number of **`invokevirtual`** executed by the `<a>` thread, and `<e>` with the number of **`invokedynamic`** executed by the `<a>` thread.

Example Output:

```
Thread: Eärnur - #static: 1765 - #special: 559 - #virtual: 2013 - #dynamic: 8
Thread: Ondoher - #static: 1566 - #special: 455 - #virtual: 1849 - #dynamic: 6
Thread: Narmacil II - #static: 1393 - #special: 385 - #virtual: 1723 - #dynamic: 4
Thread: Elessar - #static: 1798 - #special: 551 - #virtual: 1987 - #dynamic: 9
Thread: Calimehtar - #static: 1438 - #special: 415 - #virtual: 1748 - #dynamic: 5
Thread: Minardil - #static: 0 - #special: 0 - #virtual: 0 - #dynamic: 0
Thread: Telemnar - #static: 1252 - #special: 291 - #virtual: 1631 - #dynamic: 1
Thread: Telumehtar Umbardacil - #static: 1300 - #special: 347 - #virtual: 1649 -
#dynamic: 3
Thread: Eärnil II - #static: 1616 - #special: 481 - #virtual: 1869 - #dynamic: 7
Thread: Tarondor - #static: 1332 - #special: 323 - #virtual: 1707 - #dynamic: 2
```

Note 1: A correct implementation should print numbers similar to the ones above (actual numbers printed and line order can vary due to non-determinism).

Note 2: The profiler **must** maintain per-thread counters, not single, global counters, for each event of interest.

Exercise 3 (5 points)

Implement a profiler that detects all implicit lock acquisitions occurring in class `ex3.MainThread`. For each object used as a monitor (i.e., whose implicit lock is acquired at least once), the profiler must track the number of acquisitions. At the end of the application, the profiler must print the values in the following format:

```
<a> - <b> - #Locks: <c>
```

Replace `<a>` with the hashcode of the monitor, `<b>` with the fully qualified class name of the monitor, and `<c>` with the number of lock acquisitions on this monitor.

Example output:

```
924531043 - ex3.MainThread$FloatCounter - #Locks: 90
1997651848 - ex3.MainThread - #Locks: 5
1595953398 - java.lang.Class - #Locks: 135
1424087260 - ex3.MainThread - #Locks: 2
434346874 - ex3.MainThread - #Locks: 4
245199122 - ex3.MainThread - #Locks: 6
342608557 - ex3.MainThread - #Locks: 8
117920428 - ex3.MainThread - #Locks: 3
1660325198 - ex3.MainThread - #Locks: 7
1664403052 - ex3.MainThread - #Locks: 1
393087219 - ex3.MainThread - #Locks: 9
1984833688 - ex3.MainThread$DoubleCounter - #Locks: 90
```

Note: A correct implementation should print numbers similar to the ones above (actual hashcodes and line order can vary due to non-determinism).

Exercise 4 (5 points)

Implement a profiler that detects all the exceptions thrown. The profiler should store, for each different exception class, the fully qualified class name, the number of times it has been thrown, and the average calling-context depth (i.e., the number of open stack frames when the exception occurs). The values must be printed at the end of application execution, following the format below:

```
Exception class: <a> - occurrences: <b> - avg calling context depth: <c>
```

Replace `<a>` with the fully qualified class name of the exception, `<b>` with the number of times that the exception `<a>` has been thrown, and `<c>` with the average calling-context depth of exception `<a>`.

Example Output:

```
Exception class: ex4.MainThread$EvenException - occurrences: 9 - avg calling context depth: 16.33
Exception class: ex4.MainThread$OddException - occurrences: 10 - avg calling context depth: 16.80
```

Note: A correct implementation should print numbers similar to the ones above (actual numbers printed and line order can vary due to non-determinism).

Exercise 5 (5 points)

Solve this exercise with the help of the java disassembler (javap). Implement a profiler that helps you understand the behavior of method `foo` defined in `ex5.MainThread`, tracking all inputs and returned values upon each invocation of this method.

At the end of the application execution, the profiler should print the following information, as many times as `foo` was invoked:
`Input: <a> - Output: <b>`

Replace `<a>` with the value(s) of the parameter(s) passed to the method and `<b>` with the corresponding return value of that invocation.

Please state what the method computes. You can add your explanation as comments in the profiler.

Note: It is not required that this profiler is thread-safe.

Exercise 6 (5 points)

Solve this exercise with the help of the java disassembler (javap). The target application allocates a single array of Strings. Strings can be either “Valid” or “Invalid”. Method `foo` of `ex6.MainThread` checks the validity of several (but not necessarily all) Strings, by calling `String.equals(Object s)`.

Implement a profiler that tracks:

- The length of the single array of Strings
- Each string tested, together with its validity (either "Valid" or "Not valid").

At the end of the execution, the profiler should print the following output (the second line must be repeated for each string checked):

```
Length of String array: <a>
String: <b> - Result: <c>
```

Replace <a> with the length of the string array, <b> with the string whose validity is checked and <c> with either "Valid" or "Non valid", depending on <b>.

Example Output (Partial):

```
Length of String array: 108
String: ecc58b53-5524-4e78-a24d-f573e57f7d8e - Result: Not valid
String: d7f39b73-0b59-41cb-b153-9ceffd10d60f - Result: Not valid
String: 231e2abb-907d-48ad-a0fa-a3cd8ea3b99c - Result: Not valid
String: 91819a3c-7cbd-4e8c-a3ea-be3354549f82 - Result: Valid
String: 8d37aab5-12f5-413b-9355-55af3f7d3da6 - Result: Not valid
…
```

Note 1: The program is randomized and therefore the length of the String array and the number of valid strings may vary.

Note 2: It is not required that this profiler is thread-safe.

Note 3: It is not required to print the results in the order in which the strings were checked.

Exercise 7 (10 points)

Implement a profiler that detects all single-dimension array allocations, determining---for each encountered component type and array length---the number of allocated arrays.

At the end of the application execution, the profiler should print, **in sorted order**, the number of allocations, following the format below:

`<component-type>, <length>: <n> allocations`

Replace `<n>` with the number of allocations for arrays of type `<component-type>` and length `<length>`.

For example, if the applications allocates:

- 3 arrays of Strings of length 10 (`String[10]`);
- 20 arrays of Strings of length 2 (`String[2]`);
- 2 arrays of Object of length 4 (`Object[4]`);
- 2 arrays of int of length 20 (`int[20]`);

The profiler should print:

```
int, 20: 2 allocations
java.lang.Object, 4: 2 allocations
java.lang.String, 2: 20 allocations
java.lang.String, 10: 3 allocations
```

Note: Sorting should be performed as follows:

- First, by fully qualified name of the component type, alphabetically.
- Second, by array length, in numerical ascending order.

Example Output (Partial):

```
byte, 0: 5 allocations
byte, 1: 45 allocations
byte, 2: 20 allocations
byte, 3: 34 allocations
…
ex7.MainThread$A, 10: 10 allocations
ex7.MainThread$B, 4: 10 allocations
ex7.MainThread$C, 0: 10 allocations
ex7.MainThread$C, 20: 10 allocations
int, 0: 90 allocations
int, 1: 3 allocations
int, 2: 4 allocations
…
java.lang.Class, 1: 228 allocations
java.lang.Class, 2: 26 allocations
…
```

```
java.lang.Object, 0: 2 allocations
java.lang.Object, 1: 139 allocations
java.lang.Object, 2: 27 allocations
…
```

Note: Due to non-determinism, a correct implementation may print numbers different to the ones above.

Bonus Exercise (10 bonus points)

Extend the profiler implemented in Exercise 7, considering **also** allocations of multidimensional arrays. For simplicity, you may consider allocations of arrays with <= 4 dimensions.
At the end of the application execution, the profiler should print, **in sorted order**, the allocations following the format below:

```
<component-type>, <length1>[, <length2>][, <length3>][, <length4>]: <n> allocations
```

Replace `<n>` with the number of allocations for arrays of type `<component-type>` and dimension `<length1>, <length2>, <length3>, <length4>.`

For example, if the applications allocates:

- 3 arrays of String[10];
- 20 arrays of String[2][30];
- 2 arrays of Object[4][3][2];
- 2 arrays of int[20][30][40][50];

The profiler should print:

```
int, 20, 30, 40, 50: 2 allocations
java.lang.Object, 4, 3, 2: 2 allocations
java.lang.String, 10: 3 allocations
java.lang.String, 2, 30: 20 allocations
```

Note: Sorting should be performed as follows:

- First, by fully qualified name of the component type, alphabetically.
- Second, by number of dimensions, in numerical ascending order.
- Third, by length of each dimension, in numerical ascending order.

Example Output (Partial):
The output should be similar to the one of Exercise 7. but the allocations related to `ex7.MainThread` should now contain:

```
…
ex7.MainThread$A, 10: 10 allocations
ex7.MainThread$A, 10, 20: 10 allocations
ex7.MainThread$A, 12, 43: 10 allocations
ex7.MainThread$A, 50, 60: 30 allocations
ex7.MainThread$B, 4: 10 allocations
ex7.MainThread$B, 4, 5, 6: 20 allocations
ex7.MainThread$B, 7, 8, 9: 10 allocations
ex7.MainThread$C, 0: 10 allocations
ex7.MainThread$C, 20: 10 allocations
ex7.MainThread$C, 1, 2, 3, 4: 30 allocations
ex7.MainThread$C, 4, 3, 2, 1: 10 allocations
…
```