JVM & tools

Exercise 1

Using `NetBeans` create a simple Java program that takes some strings from the command line, sorts them using the method `Arrays.sort` and prints the ones whose length is even.

- **Goal**: Checking that the `NetBeans` installation works; starting simple Java coding.
- **Expected output**: Simple Java class working as described.

Exercise 2

Check the differences between files `StrangeOne.java` and `StrangeTwo.java`. Compile them, disassemble the obtained classes using `javap -c` and inspect the *byte-code* of the method `sum`. Is the *byte-code* the same? Can you explain why?

- **Goal**: Using `javap` for disassembling Java code; inspecting simple bytecodes.
- **Expected output**: One short sentence answering the two questions.

Exercise 3

Run the program `WrongQueue` and inspect its behaviour using `visualvm`. Can you explain the continuos growth of the heap? Find the code causing the bug and fix it.

- **Goal**: Using `visualvm` to inspect the memory consumed by a Java program; Reviewing Java code to detect non-trivial errors; Fixing bugs
- **Expected output**: One sentence identifying the bug in the code; A revised version of the class with minimal changes to fix the bug.

Exercise 4

The file `Strings.java` contains classes `String1` and `String2`, that just differ for the way they print the same string in a loop. At the end of the last lecture we inspected the bytecode of the `print/println` statements, identifying a difference in the allocation of memory.

Run and inspect classes `String1` and `String2` with `visualvm`, comparing in particular the allocation patterns on the heap and the time spent for Garbage Collection. Are they similar or not? Can you explain why? (Hint: inspect the relevant `PrintStream.java`'s `println` source code; you can browse it also in `NetBeans`.)

- **Goal**: Browsing the Java API source code to identify the resources used by the runtime support during execution.
- **Expected output**: One short answer to the first question, supported by a detailed explanation.

Exercise 5

Run and inspect the program `GCstrange` using `visualvm`, in particular check the evolution of the heap, the activity of the GC and the activity of the `Finalizer` thread. Using `GCstrange` as a template, write a simple class that overrides the method `finalize` in order to count how many times the garbage collector is invoked. Write a `main` to test this class.

- **Goal**: Understanding the *finalization* of objects in Java; exploiting the `finalize` method to infer simple properties of garbage collection.
- **Expected output**: A clear explanation of why the occupation of the memory increases during execution of the given program; a working Java program implementing the given specification.

Exercise 6

The classes `ASuper` and `BSub` are those of the second question of the Entry Test. Consider the files `MainA.java`, `MainB.java` and `MainC.java`: just by inspecting their code, can you say for each of them if it compiles or not? If yes, can you predict what the code prints in each case? Verify your guesses by actually compiling and executing the code.

- **Goal**: Most students provided only partly correct answers to the second question of the entry test. This exercise gives them the possibility of checking their answers.

Date: 2018-09-30

Authors: Matteo Busi and Andrea Corradini

Org version 7.9.3f with Emacs version 24