OpenJDK Hackathon 2024

Brief rules, general hints, and other things that are good to know

- Duke is the name of the OpenJDK mascot.
- The assignments in this document are to be solved on-site, at the contest location within the contest time, 17:30 – 20:30.
- Each competing team should consist of at most two people.
- As soon as an assignment is solved, let the contest staff know so that they can record your points. Note that different assignments scores different points.
- The team with the highest score collected at the end of the competition win. If two teams have the same score, the team who reached that score first will win.
- See eligibility and complete rules on separate note.
- The JVM's primary input is Java bytecode files. These are loaded, parsed and finally executed by the JVM. Bytecode files are produced from the Java source code by `javac`. You can show the produced bytecode using `javap`. Run `man javap` to learn how to use it to figure out what's going on in your compiled class file.
- `java`, `javac`, `javap`, and many other commands are available in your build directory after you have built your JDK. Remember that the changes you make in the source code will only affect the JDK you build, not other Java installations on your system.
- You can use `make hotspot` to build only the HotSpot part of the codebase and `make images` to build the whole JDK.
- Remember that the JIT compiler can optimize out code that you thought were there. When you run your Java program, you can use `-Xint` to ensure no JIT compilation takes place.
- The unix tools `find`, `grep`, and the debugger `gdb` are your friends. Use them, and don't be afraid to use Google to find tutorials and documentation!

1 point: *Vacation mode*

Duke has been running Java for more than 29 years and has decided to take a vacation to be all rested and alert for next year's 30-year anniversary. Help Duke to put the JDK in vacation mode before he leaves.

Create a new boolean JVM flag `-XX:+Vacation` that defaults to true. To verify and demonstrate your solution use `-XX:+PrintFlagsFinal` and see that the flag exists and has the correct value. Also demonstrate that the flag can be disabled using `-XX:-Vacation`.

1 point: *Mode: Vacation*

Verifying what version of the JDK you are using is easily done using the flag `-version`. The version string from your JDK will look similar to this:

```
openjdk version "24-internal" 2025-03-18
OpenJDK Runtime Environment (slowdebug build 24-internal-adhoc.vscode.jdk-uplang)
OpenJDK 64-Bit Server VM (slowdebug build 24-internal-adhoc.vscode.jdk-uplang, mixed mode)
```

The third line tells you what kind of JVM you are using. Make sure that this version string also includes the text “vacation mode” if your JVM is run with `-XX:+Vacation`. Your new output should look similar to this:

```
openjdk version "24-internal" 2025-03-18
OpenJDK Runtime Environment (slowdebug build 24-internal-adhoc.vscode.jdk-uplang)
OpenJDK 64-Bit Server VM (slowdebug build 24-internal-adhoc.vscode.jdk-uplang, mixed mode, vacation mode)
```

Demonstrate your solution by showing the code change and running your modified JVM with and without `-XX:+Vacation`. It’s ok if your version string doesn’t contain other mode strings besides your new vacation mode. After all, who cares about other modes during vacation?

1 point: *A vacation to be logged*

When working with JVM development, debugging is easily done using logging. The HotSpot virtual machine comes with its own logging framework called Unified Logging, or UL. You should use this for logging throughout this hackathon.

UL has different “tags” used to identify logging output from various sources. You can use these tags to select what kind of logging you want to enable. Duke now asks you to define a new log tag called “vacation” and use this log tag in a message which reads "Duke is on vacation, please be nice." which should be sent during VM startup if the JDK is in vacation mode.

You can read more about UL at `https://openjdk.org/guide/#logging`.

Demonstrate your solution by showing the code changes and running the JVM showing the log message when in vacation mode.

2 points: *Didn’t you get the P(al)M?*

To make sure everyone knows he’s on vacation, Duke wants to spread the word. Help him by appending a vacation Unicode character (🌴) to all strings printed by `System.out.println()`.

Note: Even though Duke did ask us to add the palm tree to all strings printed, we will only do this for strings that contains the word “Duke” since the JDK uses Java strings in many contexts internally and unexpected palm trees will cause more harm than joy.

Demonstrate your solution by showing the modified code and running a Java program that prints strings to show the added palm tree symbol.

***3 points:** Closed for vacation*

JCMD is a tool used to monitor and communicate with a running JVM from a remote process. It uses the dcmd (diagnostic command) interface to do this. Since Duke will be out, he wants to have an autoreply message in place so that anyone who tries to use a dcmd knows that he won't be around to take their call.

Make it so that all dcmd calls simply print the string "Closed for vacation" if the VM is in vacation mode. Demonstrate your solution by running `jcmd` to call into your modified JVM.

***3 points:** A safe point to nap*

Duke knows the value of a nap and will certainly have many on his vacation. He wants to implement a "Siesta feature" to avoid the rest of OpenJDK to become jealous of his resting prowess. A safepoint is a special time slice, during which all Java threads are paused and the JVM can perform garbage collection and other tasks which may affect the Java heap. Duke figures that this is a perfect time for the JVM to get some rest as well.

Help Duke to add a UL log message saying "Siesta time! (˘ʖ̲˘)" and a short sleep (a few milliseconds is a long time for a computer) to all safepoints that are executed.

Demonstrate your solution by showing the code change and by running your modified JVM showing the messages on the vacation log.

***3 points:** Less queueing, more vacationing*

Duke realized the ticket system on the airport is implemented in Java and wants to advance faster in the ticket queue. His queue number is currently 4711. The `Integer.parseInt()` method converts a String to an int. Modify it so that, when parsing the string "4711" it returns the number 1.

Also add a new test under `test/jdk/java/lang/Integer` that verifies this new behavior. To read more about running tests see `https://openjdk.org/guide/#jtreg`.

Demonstrate your solution by showing your code change and new test. Also show that your test passes with your change and fails without it.

4 points: *Packing for performance*

Duke is packing for his vacation and wants to ensure all pending jobs in his JVM can run as efficiently as possible. The JVM includes two JIT compilers: C1 and C2. The first is more lightweight and consumes less resources, while the latter is a highly optimizing compiler. To obtain fast startup and good long-term performance, a mix of C1 and C2 is used in what is known as "tiered compilation".

Help Duke analyze tiered compilation and understand its performance benefits by adding new logging to show which compilation type is selected, so that the tier level is printed whenever a method is compiled. Write a simple "Hello, world!" program and benchmark (`time`) it with tiered compilation both enabled and disabled, checking the compilation level using the newly added logging.

To demonstrate your solution, show your changes and answer these two questions:

- What runs faster?
- If tiered compilation is disabled, which is the default compiler?

4 points: *Color painting*

Duke loves to paint, and he wants to share this love with the developer community of OpenJDK. He happens to know that ZGC has a concept of "colored pointers", where certain bits of pointers to objects in Java are set and interpreted as metadata about those pointers. He was hoping that a new color can be added to all object pointers which indicate that he’s on vacation and <3 painting. Can you help him out by showing Duke where and how the code should be changed?

To demonstrate your solution, it's sufficient to show the changed code using `diff`. Don’t run the modified JVM as actually changing these bits will cause the rest of the JVM to be surprised and crash.

5 points: *The lost city of Z*

During his vacation Duke has decided to go on a road trip to search for "The lost city of Z(GC)".
Duke has a terrible sense of direction though and knows he can only find the city with the aid of a map. In the car, there is a map-book containing way too many pages to count. To find the right page containing the map to Z, he should look for clues in the virtual to physical memory mappings in ZGC. Duke has the feeling that once he sees what pages are using what virtual memory mappings, he can identify the correct page instantly!

Help Duke find his way to the lost city of Z by finding out what virtual address pages are mapped in ZGC and print the address of these pages.

To demonstrate the solution, show your code changes and run any java program with your modification(s) using the flag `-Xms1M`, which forces memory to be mapped more often (and thus increasing the chance of finding the map to Z!). Remember that Z isn’t the default GC in HotSpot, and that pages are mapped when they are allocated.

5 points: *Phase tracing*

The JVM includes two compilers: C1 and C2. The latter is a highly optimizing compiler. Such compilers are composed by multiple phases (sometimes also "passes"). Duke is planning a special vacation for each of the phases in C2 and would like to have a list of all passes that runs. Can you help him find a technique he can use for this purpose?

Use `gdb` and its breakpoint functionality to break execution on each C2 compilation phase. Write a Java program and execute it under `gdb` with your breakpoints attached and see that it stops on each compilation phase.

`gdb --args jdk/build/yourbuild/jdk/bin/java` will start `gdb`, then use `run` to start executing the JVM. Doing this a first time will load all symbols into `gdb`, then you can simply attach the breakpoints and use `run` again to run it another time with the breakpoints attached. It's normal for `gdb` to report SIGSEGVs when debugging the JDK, just continue (`c`) past them.

Demonstrate your solution by running your Java program in `gdb` and show that it hits the breakpoints.

8 points: *Getting away with it*

Duke knows that the key to a good vacation is knowing what you can get away with. He is currently bored and has finished his last crossword. To make something happen Duke decides to sneak in a bug in the JVM and is determined to do it without getting caught. Can you help him?

Introduce a bug in the class loader such that none of the tests catch it. Then, add a new test that catch your new bug.

To demonstrate your solution:

1. Show the newly introduced bug in the source code.
2. Run the `gtest:ClassLoader` tests, and show that all of them pass.
3. Add your new test in `gtest:ClassLoader` and show that it fails.
4. Fix (remove) your bug and show that your new test now passes.

At `https://openjdk.org/guide/#gtest` you can read more about how to write and run GTests in HotSpot.