OpenJDK Hackathon @ Omegapoint

General hints and things that are good to know

- Duke is the name of the OpenJDK mascot.
- 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 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.
- Inside the JVM you can use `tty->print_cr` to print to stdout.
- When working with assembly code, think about which CPU architecture you are using. What would happen (or not happen) if you change the assembly of a CPU architecture that you're not running on?
- 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:__ The saga begins

Duke is tired of misbehaving programs trying to ruin his beautiful JVM. He has therefore decided to become the official Sheriff of Java. His mission is to hunt down these bad programs by implementing special features in HotSpot. Duke is now asking for your help. He needs a way to inspect what the JVM is doing.

Your first challenge, if you choose to accept it, is to find where the JVM starts and place a gdb breakpoint on the first line of code. Being a library, the exact start point is somewhat fuzzy though, but, I mean, there is a place called create_vm() …

Use `gdb --args jdk/build/yourbuild/jdk/bin/java` to start gdb and use `run` to start executing the JVM. Doing this the 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.

OpenJDK Hackathon @ Omegapoint

---

1 point: *Arguments are futile*

Using your new breakpoint, Duke now wants to inspect the JVM arguments. Help him figure out how to do that.

---

1 point: *A new sheriff*

Next, Duke needs a way to see what classes are loaded into the JVM. The Unified Logging (UL) framework is used throughout the JVM to report various events. It is heavily used by JVM developers for debugging and benchmarking.

Write a Java program with a class called DukeTheSheriff and show using UL that it is loaded into the JVM.

---

2 points: *Inspecting Gadgets*

Duke knows that the JVM has special timeslices, called safepoints, during which all Java threads are paused. This is where the JVM can perform garbage collection and other tasks which may affect the Java heap. Duke wants to use this opportunity to inspect the running application. Help Duke out by printing "In the name of Java, halt! Let me inspect your goods." to stdout each time a safepoint begins. Write and run a Java program which shows this working.

---

3 points: *Message in a byte code*

The JVM may execute Java bytecode both inside of an interpreter and as part of native compiled code. The interpreter is implemented by providing assembly code snippets for each bytecode and executing this assembly directly. Help Duke spread his vision by finding the code responsible for generating the iconst_2 bytecode instructions and alter it to also print "Omegapoint!" to stdout. Write and run a Java program which shows this working.

Hints

- The relevant source code generates assembly code, is that architecture agnostic or architecture specific code? Where in the source tree can you expect such code to be found?

---

3 points: *Stop the Stream!*

Duke also wants to make sure that no one prints any mean Strings about him. To achieve this, he asks you to make sure that the call System.out.println() simply refuses to print any Stings containing the word “Duke”, and instead prints “Censored” if such a String is encountered.

Demonstrate with a small Java program that your solution works.

---

4 points: *In a class of his own*

Duke's attempt to take charge of Java has been noticed on various social media platforms and people have started producing methods named "dukeIsNotMySheriff" in protest. Duke is very unhappy with this and wants to detect when a class with a method named "dukeIsNotMySheriff" is being loaded by the JVM. When such a class is detected, the JVM should print "Criminal activity detected! You do not deserve to execute any more Java." and exit.

The classfile produced by this program should fail to execute:

```
public class Test {
    public void dukeIsNotMySheriff() { }
    public static void main(String[] args) { }
}
```

---

5 points: *A dreaded diagnose*

To further suppress the resistance, Duke decides to ban certain features that could be used to gain control of the JVM.

In the file `diagnosticCommand.cpp` add this line at the very beginning of `HelpDCmd::execute()`:

```
void HelpDCmd::execute(DCmdSource source, TRAPS) {
   output()->print_cr("This is a forbidden feature!");      ← Add this
```

What feature is it that Duke wants to stop the resistance from using? Show a successful execution where the new text is printed.

Hints

- In order to gain control over a JVM you would need a long running JVM to take control over. Thread.sleep() is a useful call.

OpenJDK Hackathon @ Omegapoint

5 points: *A secret message*

Duke needs a secret way to communicate with his allies. Help him implement two static methods in the String class. The first should translate a given text to the robber's language ("Rövarspråket"), and the second translates the secret message back to normal text. The second method should fail gracefully (throwing an Exception) if the given text is not in the correct format.

6 points: *Trace the Phase*

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"). To plan his next move Duke now needs to know which compiler phases are present in his build. Unfortunately, the resistance has locked away his source code. He now ask you to find a technique he can use for this purpose. You, being a clever JVM developer, figure that he can use gdb and breakpoints in order to see which phases are being run.

Your task is therefore: 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 show that it breaks on each compilation phase.

10 points: *The final collection*

Some time has passed since The Great Duke Conflict began and two factions have emerged:

- The Pro-Duke fraction, who seeks to have Duke be Benevolent Dictator for Life of OpenJDK.
- The Anti-Duke fraction, who seeks to have an OpenJDK based on democracy and peer review.

Because of your earlier engagements you found yourself on the Pro-Duke side, but after Duke "forgot" to take out the trash (even though the schedule said it was his turn!) one too many times you've switched sides.

While investigating how to take revenge on Duke for his non-garbage collecting ways you've discovered that allies of Duke generate a secret signature in their programs' object graphs to show their allegiance to their leader: They make objects that spell out DUKE in Morse Code.

(The saga continues on the next page…)

OpenJDK Hackathon @ Omegapoint

You know that a traditional tracing garbage collector consists of two phases:

1. Marking, where the GC traverses from the root set (the stack and global variables, typically) into all reachable objects in a depth-first fashion.
2. Sweeping, where the GC cleans up all unreachable objects (since these are now impossible to use).

Perhaps one of these phases allow you to take down programs that generates the secret morse code message DUKE?

You decide on the following plan of attack:

Given these classes:

```
class Morse { }

class Long extends Morse {
    Morse next;
    Long(Morse n) {
        super();
        this.next = n;
    }
}

class Short extends Morse {
    Morse next;
    Short(Morse n) {
        super();
        this.next = n;
    }
}

class End extends Morse { }
```

You want to change the Serial Garbage Collector such that it identifies the Morse object sequence that spells out "DUKE", print "Down with Duke! Direct Democracy forever!" and exits the VM. Write an example Java program that when run through your new JVM shows this functionality.

Hints

- First write your secret message program.
- Use `javap` to ensure that `javac` doesn't optimize out the object sequence you create.
- Use `-XX:+UseSerialGC` to use the Serial Garbage Collector.