Java 9: JShell, Microbenchmarking and the Module System

David Gretzschel, 359043
Lukas Schneider, 348401

SS 2017

1 Introduction

This paper is introducing three of the new features coming with Java 9.
First, we present the JShell, a command line environment that can directly evaluate short snippets of Java code. We will look at the possibilities and limits of JShell as a way to create small projects and as a tool to learn the behaviour and nuances of the Java language.
Next we consider JMH, a benchmarking framework that can be used to measure and compare the performance of of parts of your code/the performance. This feature is interesting because performing a benchmark manually is not trivial - anything from optimizations to garbage collectors can skew the result. JMH, having been created by the JVM developers themselves, may make meaningful microbenchmarking possible and feasible to people without extremely deep JVM knowledge.
Finally, we will take a look at the Java Platform Module System, which started out as Project Jigsaw almost ten years ago. We will look at how modules will work in Java 9 and which existing problems they can and can't solve.

2 JShell

The Java Shell (short: JShell) is a program that allows you to test short pieces of Java code (referred to as 'snippets') right inside a terminal environment, without having to write a .java-file, compile it to a .class and having to manually execute it. With a bit of practice it can also serve as a fairly efficient tool to create smaller projects with multiple classes as well. Language interpreters, that run in a terminal environment like this are generally known as REPL (Read-Eval-Print-Loop) and help users to create small applications, quickly test if a function does what it should, help exploring the methods of an unfamiliar API, analyse algorithms and through the quick feedback of an action, generally serve as a valuable learning tool. Unix Shells and many scripting languages for example Bash also run in a such an environment, making them easy to pick up.

Other higher level programming languages like Python with the Python Interpreter, Scala with the Scala REPL have already implemented this idea. For Java specifically this has already been done by the Crash project[1] or the BeanSHell[2] project, however till Java 9, there has not so far been a Shell officially supported by Oracle itself, as a part of the Oracle-JDK.

In the popular NetBeans IDE the JShell has already been integrated in the Nightly version, but on any Linux system, that has JDK9 installed, or from any Windows machine, for that matter, the JShell can be invoked from Bash, the command prompt or from

Powershell by simply typing *jshell*. Here, we are showing how to do some common Java-related programming tasks with it:

```
jshell> int number = 1
number ==> 1
jshell> 5
$1 ==> 5
```

The first example declares the variable number of type int and initializes the value with 5. In the second example, the expression result is automatically assigned to a variable `$1` of an automatically determined type. Note that the semicolon at the end of a statement can be omitted.

```
jshell> double foo(int x){double y=x*5.0; return y;}
|  created method foo(int)

jshell> String foo2(String a){
   ...> String b = a+a;
   ...> return "some return value";
   ...> }
|  created method foo2(String)

jshell> foo(5)
$10 ==> 25.0
```

Here, a method `foo` is created with its full body written in one line. It is possible to write the method code across multiple lines as demonstrated with `foo2`. If a line is terminated before the code block is closed with `}`, JShell prompts for the next line until the method body is closed. Methods can be called in the usual way, their return value is again saved as a variable if it is not caught.

```
jshell> public class Rectangle{
   ...> public Rectangle(double length, double width){
   ...> this.length = length; this.width = width;
   ...> }
   ...> public static void howDoIcreateYou(){
   ...> System.out.println("Simply call my constructor Rectangle with two
    values");
   ...> }
   ...> private double length; private double width; double area;
   ...> public void setArea(){
   ...> this.area = length*width;
   ...> }
   ...> public void whatAreYou(){
   ...> System.out.print("I am a rectangle with an area of: "+ area + " cm
    ^2. ");
   ...> System.out.print("But my other dimensions are private");
   ...> }
   ...> }
|  created class Rectangle

jshell> Rectangle righty = new Rectangle(5.0,7.4);
righty ==> Rectangle@31a5c39e
```

Classes can be defined in the same way as methods. Their code can once again be entered across multiple lines. Objects of classes can be created in the usual way.

```
jshell> /list

   1 : int number = 1
   2 : 5
   3 : double foo(int x){double y=x*5.0; return y;}
```

```
6    ...
```

The command `/list` provides an overview over the lines of code entered so far. Using the line numbers, one can edit a snippet with the command `/edit <line>`, which opens an external text editor in another window.
If one simply enters `/edit`, all snippets so far entered are shown in the editor, which is now an ideal location to export your work. From here changes can be saved and evaluated, without leaving the editing window or without the fingers having to leave the keyboard as the user can see result of his actions immediately (e.g. class compiles without error or with a different more interesting error) in the shell. This is good for a student new to Java, as it allows them to If one does not like the built-in, fairly limited editor, it can also be changed to a different one, for example vim, with the command `/set editor vim`. If someone needs some specific Java functionalities often in his session, "import package" can also be used within the shell to import any of the SE-Edition-included packages and just typing "import" gives an overview of commonly used packages.
The usage of JShell is very simple and intuitive. Someone who is learning the (procedural) basics of the Java language can get value out of this tool, as it allows them to 'fail much faster' and get immediate error messages. However, once they want to start experimenting with classes and objects, a basic IDE will do a much better Job at assisting the learning process than JShell.

3 Java Microbenchmarking Harness

A programmer discovering that a program is running much slower than expected, often has no good immediate answer, why exactly that is. Figuring out the exact location of the slow running pieces of code, bottlenecking the performance of a program is hard, especially when working with many unfamiliar modules or reviewing code written by someone else. However, often the programmer has a suspicion of which pieces of code could be the ones responsible for the slowdown and wants to test them individually. Testing the performance of code pieces in isolation is called microbenchmarking. However, whilst simply running parts of the code one is interested in, measuring the time or the average time of many iterations, might give a seemingly reasonable result, there are several problems with such a delightfully straightforward approach. For example a single line of code creates less load for the CPU, than the whole, potentially very resource-intensive program itself, making the benchmark run in a potentially unrepresentative environment[8].

The code you want to test the speed of might actually run fast in your tests, but maybe just on your specific processor architecture, not on most others, which you might actually be more interested in[8]. When testing two implementations of a functionality in your code against each other, then code-piece 1 might be running much faster than code-piece 2 but only after some warm-up cycles, which, depending on your application, might or might not be an acceptable trade-off. Warm-up cycles here refer to the first executions of a method by the JVM, that don't yet run at maximum possible speed, because the JIT (more on that in the next paragraph) has not made all possible optimizations yet[14],[4].

Also any modern operating system optimises for a program,once it has run a few cycles, wisely preparing to provide resources for several iterations more[12].

The thing to note about Java specifically, is that the Compiler itself does not do any optimizations, but merely generates the bytecode, exactly like the potentially inexperienced programmer has written it[10]. All of Java's optimisations are done at the JVM-level (Java Virtual Machine), which uses a built-in JIT-compiler (Just In-Time). Just-in-Time compilation means, that the program is executed before all its bytecode has been translated

yet and that the JIT keeps scanning the code continuously, using sophisticated algorithms to determine, which parts would be worthwhile to compile ahead of time and then doing so. The JIT also heavily optimises the code when compiling from bytecode[17]. So even, when looking at the bytecode with $javap - c$, one cannot see, how the code is actually going to run.

A big problem for microbenchmarking is the tendency of the JVM to conscientiously discard all computations, that are never used. It considers this unnecessary, 'dead' code and simply eliminates it. Under normal circumstances, this is of course great, but isolated pieces of code are of course full of computations, that are only used by other parts, which are not included in the benchmark. This behaviour is called Dead Code Elimination. The consequence is, that e.g.

```
public class MyBenchmark {
    public void testMethod() {
        int a = 1;
        int b = 2;
        int sum = a + b;
    }

}
```

would be just run as an empty test Method, because sum is never part of any return statement or produces a side effect, line x is discarded and the parts, that make up sum, a and b are therefore also not used and their lines v,y are also discarded.

Constant Folding Constant folding is another common and in a benchmarking case, problematic JVM optimization. If a value (that is in this case 'returned' and thus not subject to Dead Code Elimination) is always calculated by adding the same two integer variables, the JVM will likely conclude it to be unnecessary to actually do the calculation, if the method is called multiple times. It will simply replace the calculation with it's result, meaning, that:

```
public int testMethod() {
        int a = 1;
        int b = 2;
        int sum = a + b;
        return sum;
    }
```

might turn into this:

```
    public int testMethod() {
        int a = 1; %strikethrough, because of dead code elimination
        int b = 2; %strikethrough, same
        int sum = 3;
        return sum;
    }
```

or even just this:

```
        public int testMethod() {
        return 3;
    }
```

So just replacing values, that would be provided from other parts of the program, with constant values, just wont be possible in a straightforward manner. However telling the JVM to not use any such optimisations would be unrealistic as well, since the JVM might use other optimisations on our code, if it can analyse it in the full context of its program.

As Brian Goetz, the author of "Java concurrency in practice" noted in 2005: "So, how do you write a perfect microbenchmark? First, write a good optimizing JIT. Meet the

people who have written other good, optimizing JITs (they're easy to find, because not too many good, optimizing JITs exist!). Have them over to dinner, and swap stories of performance tricks on how to run Java bytecode as fast as possible. Read the hundreds of papers on optimizing the execution of Java code, and write a few. You will then have the skills you need to write a good microbenchmark (...).[8]" Microbenchmarking is of course done by the actual architects of the Java language, in order to improve the performance of its code. however it wasn't a particularly easy thing to do for a non-expert, until the "Java Microbenchmarking Harness" was released for the OpenJDK. It will also be part of Oracle-Java 9[22]. The "Java Microbenchmarking Harness" is a tool, that allows to account for at least some of the headaches and difficulties just described, that are inherent in Micro-benchmarking.

Constructing the JMH is most reliably done by using a Maven Archetype and is well documented here[15]. Note that at the time of this writing it still requires open-jdk and will not work with the Java 9 preview, unlike JShell and the module system. A fully built testing structure should look like this:

```
├── pom.xml
├── src
│   └── main
│       └── java
│           └── com
│               └── Jon_Doe
│                   └── MyBenchmark.java
```

Figure 1: File structure for the benchmark

You put the code you want to test into the generated MyBenchmark.java-file inside the test method.

```
1 package com.jon_doe;
2 import org.openjdk.jmh.annotations.Benchmark;
3 public class MyBenchmark {
4     @Benchmark
5     public void testMethod() {
6         // This is a demo/sample template for building your JMH benchmarks.
      Edit as needed.
7         // Put your benchmark code here.
8     }
9 }
```

We can select different modes for the benchmark run by adding specific Java annotations above the test method.

Basic settings, which can all be combined with each other include Throughput-mode,Average Time, Sample Time and Single Shot Time. Throughput is the default selection and measures how often the test-method could be executed per second, Average Time measures the average time it takes to run a single method-execution. Sample Time shows the time for each execution and Single Shot time forces the benchmark to run the method only once without any warm-up iterations. The number of warm-up and measurement iterations is usually set in the java command (if Single Shot is not used) that executes the class file as execution parameters and if no parameters are given, it uses Benchmark-Mode-specific default values. When wanting to test multiple different test-methods at once, it is possible to create a runner-class file that can run multiple benchmark classes at once, in which

the iteration and Benchmark-Mode values can also set directly in the runner-class file[15]. The created benchmarks.jar is self contained, meaning, that you can copy that JAR file to another computer and run the same JMH benchmarks from there.[13]

To prevent the constant folding and dead code elimination problem it is necessary to not directly set constants at all, for the method to work with. Instead, we are using a so called state object, which is an object from a class called State, with the sole purpose of storing our needed values as public object attributes.

For the code elimination problem, we can use the descriptively named Blackhole objects, whose consume-function we use on each constant threatened by elimination. Like black holes in astronomy, their consume function is a void, so they fittingly return nothing. However, they create an unseen, inconsequential side effect, thus freeing our variables from the 'dead code'-label[13]. So our full example preserving all our computations and variables would have to be put into a benchmark like this:

```
package com.jon_doe;

import org.openjdk.jmh.annotations.*;

public class MyBenchmark {

    @State(Scope.Thread)
    public static class MyState {
        public int a = 1;
        public int b = 2;
    }


    @Benchmark
     public void testMethod(MyState state, Blackhole blackhole) {
        int sum = state.a + state.b;
        blackhole.consume(sum);
   }
}
```

Benchmarks are of course still very hard to get exactly right, but the Java Microbenchmarking Harness offers a powerful library (there are many more tools, than shown here, e.g. for dealing with inheritance, interrupts, sync iterations and many other topics[16]), documentation and a comfortable build structure, making the topic much less intimidating than before. The fact, that the compiled class-files actually running the benchmarks are self contained, means that the same benchmark can be done across multiple different systems easily and make the JMH a great one of the great new features of Java 9.

4 Java Platform Module System

4.1 Motivation

Modules, similarly to packages, can be thought of as containers that group classes and other entities together. In Java, packages can contain classes as well as other packages. A package provides a separate namespace for its classes and limits accessibility from the outside to public classes and methods. Modules provide a way to restrict access to entire packages and all its classes, whether they are public or not. On top of that a module declares its dependencies, which previously had to be done by third party module systems without support from the JVM [5].

4.2 Module Definitions

Central to a module is the file `module-info.class`, which defines [20]:

- the (unique) name of the module
- which other modules are required
- which packages are exported

It is generated from a `module-info.java` file, which looks like this:

```
module my.favourite.module{
  requires my.other.module;
  exports my.favourite.module.api;
  mein code in latex
}
```

The module name is `my.favourite.module`. Module names have to be unique, but apart from that, there are no strict naming rules. One proposed naming convention is to give a module the full name of the highest package that is contained in the module, which in turn should have a reverse-DNS name. For example, the Google Guava project should have the name `com.google.common` [9], since the common package contains all Guava sub-packages. However, this method breaks down once there is no clear highest package or if several modules use different packages from the same package [7].
In the example, `my.favourite.module` requires code from the module `my.other.module` to function. We say `my.favourite.module` **reads** `my.other.module`, or `my.other.module` is **readable** by `my.favourite.module` [19]. This means that `my.favourite.module` has access to the packages that are exported by `my.other.module`. Likewise, other modules that read `my.favourite.module` can access the public types contained in `my.favourite.module.api` [19].

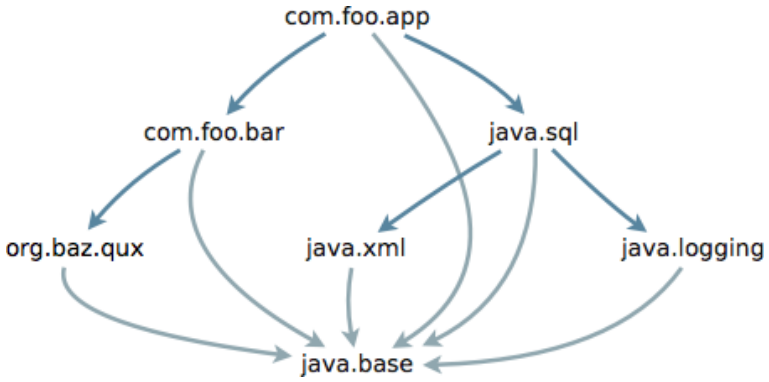

Figure 2: Exemplary module graph. Source: [19]

Module dependencies can be visualized in a graph. A directed edge from `java.sql` to `java.xml` means that `java.sql` reads `java.xml`. All the `java.` modules are part of the Modular JDK. The module `java.base` contains all core classes and packages, including, for example, `java.lang`, `java.util` and `java.math`. The edges going into `java.base` are lighter because these dependencies are not explicitly stated. By default, all modules implicitly read `java.base` [18].
The Module System does not support circular dependencies, so the module graph has to be acyclic [20]. This seems a reasonable restriction, but in practice, circular dependencies

exist in the open source world. The lack of support for circular dependencies will make it hard to modularize some projects [5].

4.3 Exemplary use of Modules

Lets assume that we have a small codebase with data structures and algorithms. Our package hierarchy looks like this:

```
com
└── proseminar
    ├── algorithms
    │   ├── search
    │   └── sort
    └── structures
        ├── internal
        ├── graphs
        ├── lists
        └── trees
```

Figure 3: Exemplary package tree

It makes sense to split up our project into two modules: structures and algorithms. Following the naming convention we should name our modules `com.proseminar.algorithms` and `com.proseminar.datastructures`.
The code in algorithms uses some of the datastructure classes, so we have a dependency of `com.proseminar.datastructures` by `com.proseminar.algorithms`.
Now let's assume that we don't want the public classes in the `internal` package to be used by other modules because we might want to modify their method signatures in future updates. Declaring these classes `private` or `protected` is not a solution since they are referenced by code in other packages. If we modularize our code, however, we can specify which packages are available for outside use.
The `module-info.java` files should look like this:

```
module com.proseminar.structures{
  exports com.proseminar.structures.trees;
  exports com.proseminar.structures.lists;
  exports com.proseminar.structures.graphs;
//  (not exported) com.proseminar.structures.internal
}
```

```
module com.proseminar.algorithms{
  requires public com.proseminar.structures;
  exports com.proseminar.algorithms.search;
  exports com.proseminar.algorithms.sort;
}
```

Here we added the `public` attribute to the dependency, which has the effect that all modules reading `com.proseminar.algorithms` also implicitly read the module `com.proseminar.structures`.
A developer including these modules into his project has to add a `require` clause to his own

module declaration. It is sufficient to only explicitly read `com.proseminar.algorithms` because the `public` attribute creates an implicit dependency of `com.proseminar.structures` as well [20].
The public attribute is useful if a method signature refers to a class from another module. An example for this is the interface `java.sql.Driver` in module `java.sql`, which declares the following method:
`public Logger getParentLogger();`
However, `java.util.logger.Logger` is part of the module `java.logging`. As we can see in Figure 2, `java.sql` reads `java.logging`. However, it does so with the public attribute:

```
module java.sql{
  requires public java.logging;
    ...
}
```

This ensures that classes in modules which read `java.sql` can call `getParentLogger()` without issue because they implicitly read `java.logger` as well [19].

4.4 The Modulepath and Compatibility

In Java 8, the location of classes and resources is specified by the classpath.
JPMS introduces the *modulepath* in addition to the classpath. When checking module dependencies, Java will look for modules on this path. To ensure compatibility, the module system treats all code on the classpath as a single special module called the *unnamed module*, which exports all of its packages and reads the entire modularized JDK as well as all modules on the modulepath. So in theory, nothing changes for developers if they decide to ignore the module system and put their code on the classpath [21].
When modularizing their projects, developers are faced with the problem that modules in the modulepath can not depend on the content of the classpath (the unnamed module). That means that if a project depends on a jar file that is not a module, simply putting it in the classpath while the project is located as a module on the modulepath will not work. The solution for this is to allow non-modular jar files (which lack a `module-info.class` file) on the modulepath and considering them to be modules with default properties. These so called *automatic modules* have the same name of the jar file and export all packages. They also read all other modules, including the unnamed module and the modularized JDK. This way, developers can declare dependencies to jar files that have not yet been modularized yet [6].
One problem with this is that the jar file names of most open source libraries differ from their their reverse-DNS module names. For example, the file name of Google Guava is `guava`, while the likely module name will be `com.google.common`. So it is possible to declare a dependency on the non-modular jar file by adding `requires guava;` to the module declaration. However, once the jar file is turned into a module it ceases to be an automatic module and can no longer be referred to by its file name [9] [6].
Additionally, if multiple modules on the module path depend on Guava, they must all refer to either `guava` or `com.google.common`. A package can only be loaded once, so adding both the modular and non-modular jar files (which contain the same packages) to the module path is not possible. The only option is to update all modules requiring `guava` at the same time, so they all require `com.google.common`. While this is not an insurmountable obstacle, it will be very hard to avoid chaos in the migration process of open source projects. A proposition to mitigate the chaos is to allow a MANIFEST.MF entry in non-modular jar files with the name of the future module name. Java could then allow dependency references to both names. It remains to be seen what the JPMS team

will do about this [6].

5 Summary

In the first chapter we explored the possibilities of the Java Shell for teaching basic Java faster and more efficiently than ever before.
Next we took a look at the difficulties and dangers of microbenchmarking and how we can use the JMH to help us work around at least some of them.
In the third chapter, we showcased the Java Module System. In particular, we explained what modules are and how they are used with a small scale example. The importance of the use of modules increases with the size of the project, as they significantly improve clarity and maintainability. However, the success of JPMS will most likely depend on how the developer team responds to the compatibility issues raised by the community.

References

[1] `http://www.crashub.org/`, (accessed May 13, 2017).

[2] `http://www.beanshell.org/intro.html`, (accessed May 13, 2017).

[3] `https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html`, (accessed May 27, 2017).

[4] H. D. Center. Warming up a java process. `https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html`, Last updated 09 March 2017 (accessed May 27, 2017).

[5] S. Colebourne. Java 9 modules - jpms basics. `http://blog.joda.org/2017/04/java-9-modules-jpms-basics.html`, 2017 (accessed May 14, 2017).

[6] S. Colebourne. Java se 9 - jpms automatic modules. `http://blog.joda.org/2017/05/java-se-9-jpms-automatic-modules.html`, 2017 (accessed May 14, 2017).

[7] S. Colebourne. Java se 9 - jpms module naming. `http://blog.joda.org/2017/04/java-se-9-jpms-module-naming.html`, 2017 (accessed May 14, 2017).

[8] B. Goetz. Java theory and practice; anatomy of a flawed microbenchmark, is there any other kind? `https://www.ibm.com/developerworks/library/j-jtp02225/`, February 22, 2005 (accessed May 27, 2017).

[9] Google. Guava: Google core libraries for java 21.0 api. `http://google.github.io/guava/releases/21.0/api/docs/`, 2010-2017 (accessed May 14, 2017).

[10] P. Haggar. Java bytecode:compiler options. `https://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/`, July 1, 2001 (accessed May 27, 2017).

[11] G. C. Hillar. *Java 9 with JShell*. Packt, 2017.

[12] in silicio. What is microbenchmarking? `http://stackoverflow.com/questions/2842695/what-is-microbenchmarking`, last edited May 17, 2010 (accessed May 27, 2017).

[13] J. Jenkov. Jmh - java microbenchmark harness. http://tutorials.jenkov.com/java-performance/jmh.html, 2015-09-16 (accessed May 27, 2017).

[14] NawaMan. answer to:technique or utility to minimize java "warm-up" time? http://stackoverflow.com/questions/1481853/technique-or-utility-to-minimize-java-warm-up-time, Sep 26,2009 (accessed May 27, 2017).

[15] openJdk. Code tools: jmh. http://openjdk.java.net/projects/code-tools/jmh/, (accessed May 27, 2017).

[16] openJdk. jmh-samples. http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/, (accessed May 27, 2017).

[17] J. Paul. Difference between jit and jvm in java - interview question. http://www.java67.com/2013/02/difference-between-jit-and-jvm-in-java.html, Feb 2013,(accessed May 27, 2017).

[18] M. Reinhold. Jep 200: The modular jdk. http://openjdk.java.net/projects/jigsaw/, 2014-2017 (accessed May 14, 2017).

[19] M. Reinhold. Project jigsaw. http://openjdk.java.net/projects/jigsaw/spec/sotms/, 2016 (accessed May 14, 2017).

[20] F. Troßbach. First steps with java 9 and project jigsaw. https://blog.codecentric.de/en/2015/11/first-steps-with-java9-jigsaw-part-1?ref=part2, 2015 (accessed May 14, 2017).

[21] F. Troßbach. First steps with java 9 and project jigsaw – part 2. https://blog.codecentric.de/en/2015/11/first-steps-with-java9-jigsaw-part-2, 2015 (accessed May 14, 2017).

[22] P. Verhas. Microbenchmarking comes to java 9. https://javax0.wordpress.com/2016/09/11/microbenchmarking-comes-to-java-9/, 11 September, 2016 (accessed May 27, 2017).