vms Software

Java 1.8 (Java 8) Update

A VSI Just-In-Time Presentation ☺

Camiel Vanderhoeven & Brett Cameron

OpenVMS Boot Camp 2016 – 10014b

26-SEP-2016

vms Software

Java 1.8 (Java 8) Update

This information contains forward looking statements and is provided solely for your convenience. While the information herein is based on our current best estimates, such information is subject to change without notice.

Agenda

- Java 8 port
  - Project outline
  - Project approach
  - Current status
- Java 7 new features
- Java 8 new features

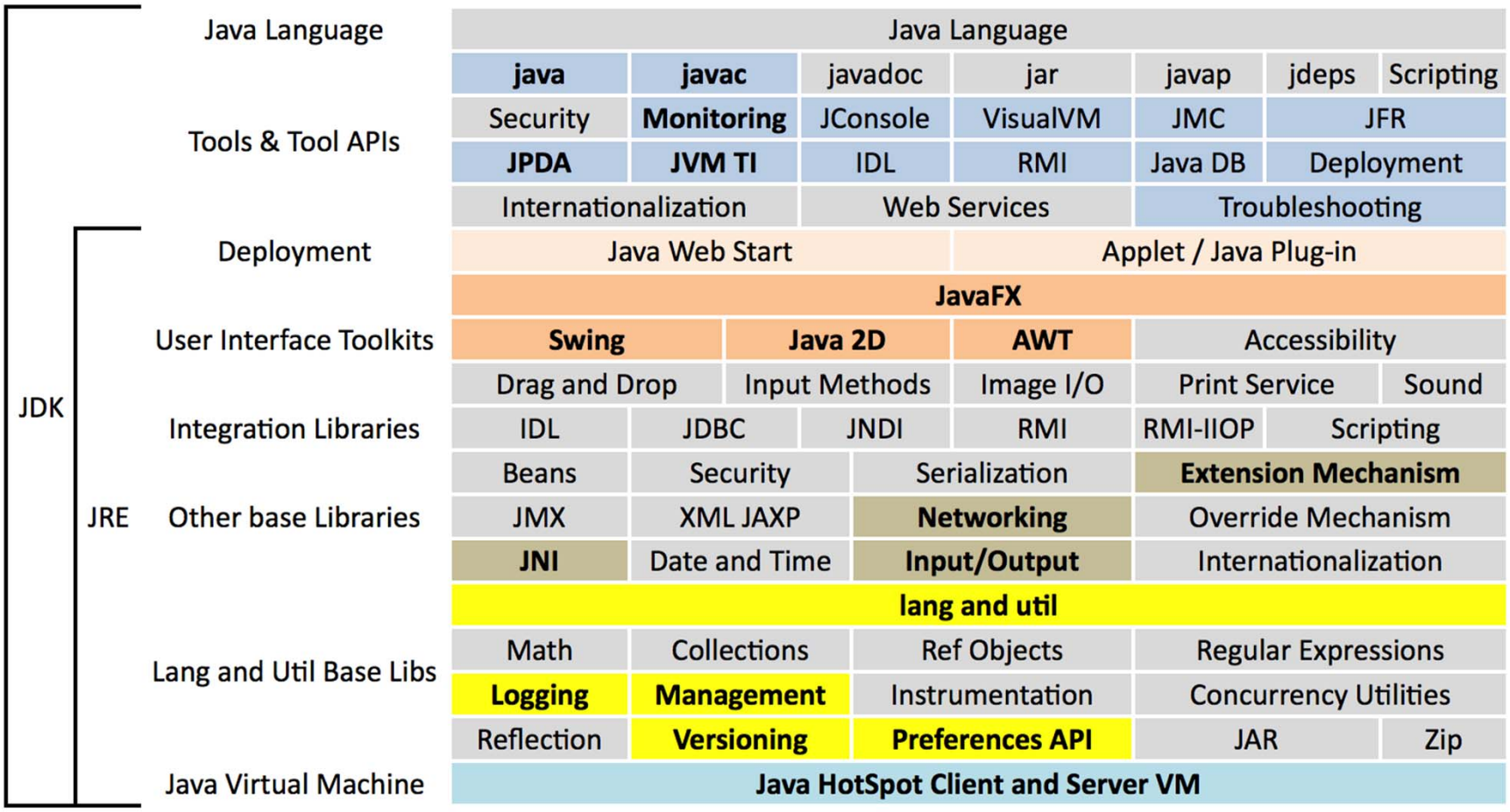
Java 8 Port : Project Outline

- Joint HPE/VSI project
- Java 1.8 for HPE and VSI OpenVMS (8.4+, IA64)
- Starting point:
  - Java 1.6 port to OpenVMS
  - Java 1.8 port to HP-UX
- Updating a number of Java applications as part of the port: Tomcat, Axis2, Ant, Wsit, …
- Probably around 90% complete (depending on how you measure)
- Field test planned for calendar Q4 2016
- Release planned for calendar Q1 2017

Java 8 Port : Project Outline

- Large amount of work
  - HotSpot:
    - 99% C++
    - 2,836 source files in 58 directories
    - 1,383,700 lines of code
    - 15% of code is very CPU/OS specific
    - ~2,500 code changes required
  - JDK:
    - 83% Java
    - 26,706 source files in 1,005 directories
    - 6,576,132 lines of code
    - ~1,200 code changes required

Java 8 Port : Project Outline



Java 8 Port : Project Outline

- Scope of work:
  - Port : add VMS-specific functionality
    - Part merge from Java 6 code
    - Part new development
    - Java 6 was mixed 64-bit (HotSpot) / 32-bit (JDK)
    - Java 8 is pure 64-bit
  - Test and debug
    - Ad nauseam
    - Using real Java code: clojure, ActiveMQ, Tomcat, DaCapo, …
  - Adapt Java-based applications we ship
  - Documentation

Java 8 Port : Project Approach

- Porting approach:
  - Different “rounds”
  - Round 1: Linux exploration
    - Before even seeing the Oracle/HP code
    - Determine the minimum set of changes required to turn Linux OpenJDK Java 6 into Java 8
    - End product: insight in the Java codebase, list of possible code changes to check for
  - Round 2: Applied insight
    - Selectively merge Java 6 code changes and add new code changes based on the Linux-based exploration. Harder changes left for later
    - End product: lots of “easy changes” out of the way

Java 8 Port : Project Approach

- Porting approach:
  - Round 3: hack-a-thon / codefest
    - Debug-driven development
    - Finding and fixing bugs
    - Non-systematic, based on programmers' instincts
    - End result: somewhat functional Java8, but crash-prone

This can’t be good…

```
Iteration 1: SQRT(4.0) = 2.0
Iteration 2: SQRT(4.0) = 2.0
     ...
Iteration 10028: SQRT(4.0) = 2.0
Iteration 10029: SQRT(4.0) = 2.0
Iteration 10030: SQRT(4.0) = 29.8562832991
Iteration 10031: SQRT(4.0) = 29.8562832991
Iteration 10032: SQRT(4.0) = 29.8562832991
Iteration 10033: SQRT(4.0) = 29.8562832991
     ...
```

Java 8 Port : Project Approach

- Porting approach:
  - Round 3: hack-a-thon / codefest
    - Debug-driven development
    - Finding and fixing bugs
    - Non-systematic, based on programmers' instincts
    - End result: somewhat functional Java8, but crash-prone
  - Round 4: Systematic drudgery
    - Individually reviewing every single VMS-specific line of code in the Java 6 and Java 8 code bases
    - Making reasoned corrections to the code
    - End result: functional Java8, but still with some omissions and bugs

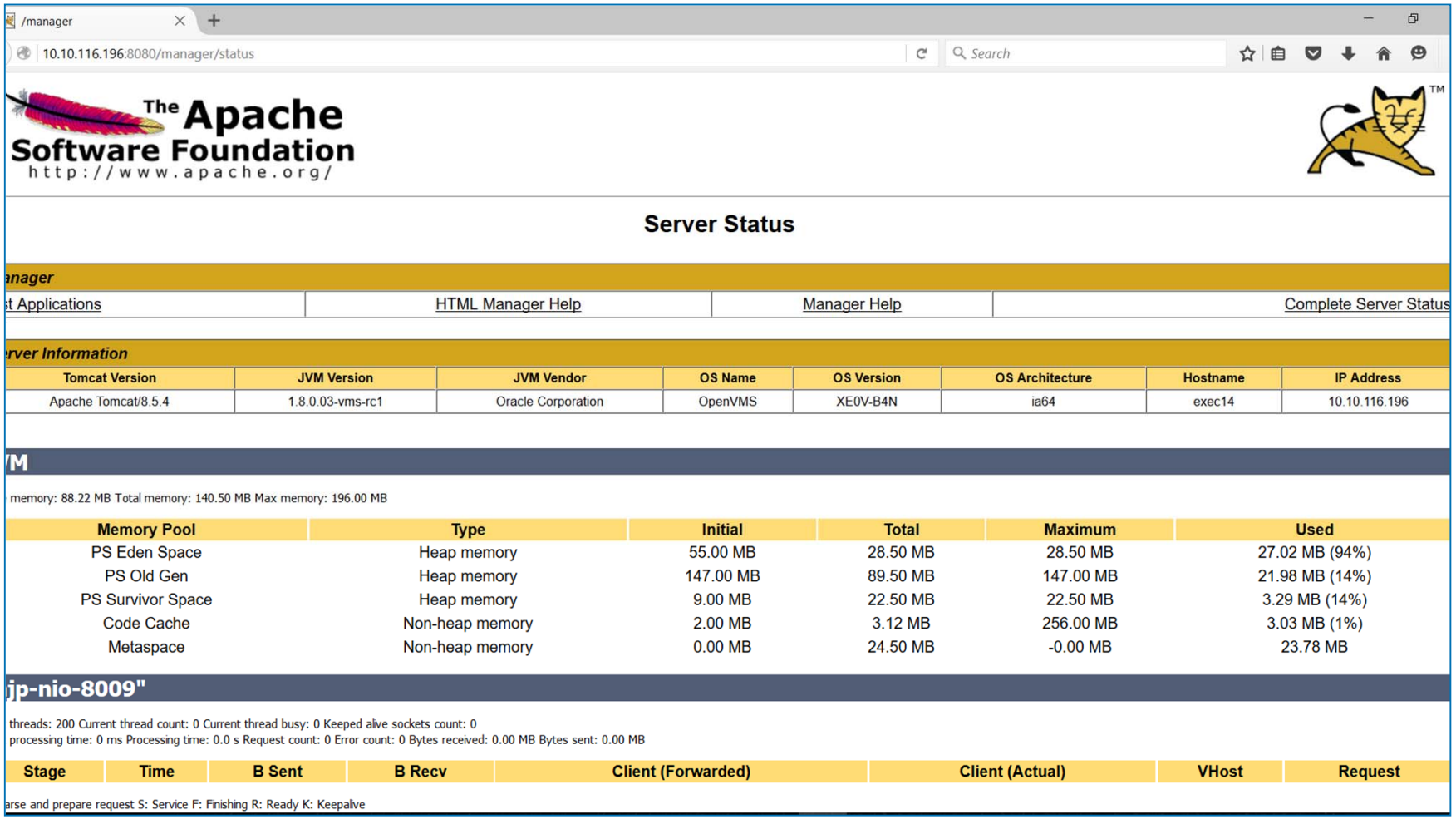
Java 8 Port : Project Approach

- Porting approach:
  - Round 5: Debugging
    - Back to debugging, still some nasty bugs out there
    - Add more applications to the test mix to find more bugs
    - End result: nearly bug-free Java 8, but with some missing bits
  - Round 6: Filling in the blanks
    - Add VMS support to new Java features (like the re-coded graphics system)
    - End result: field-test-ready Java8
  - After field test: probably another round of debugging

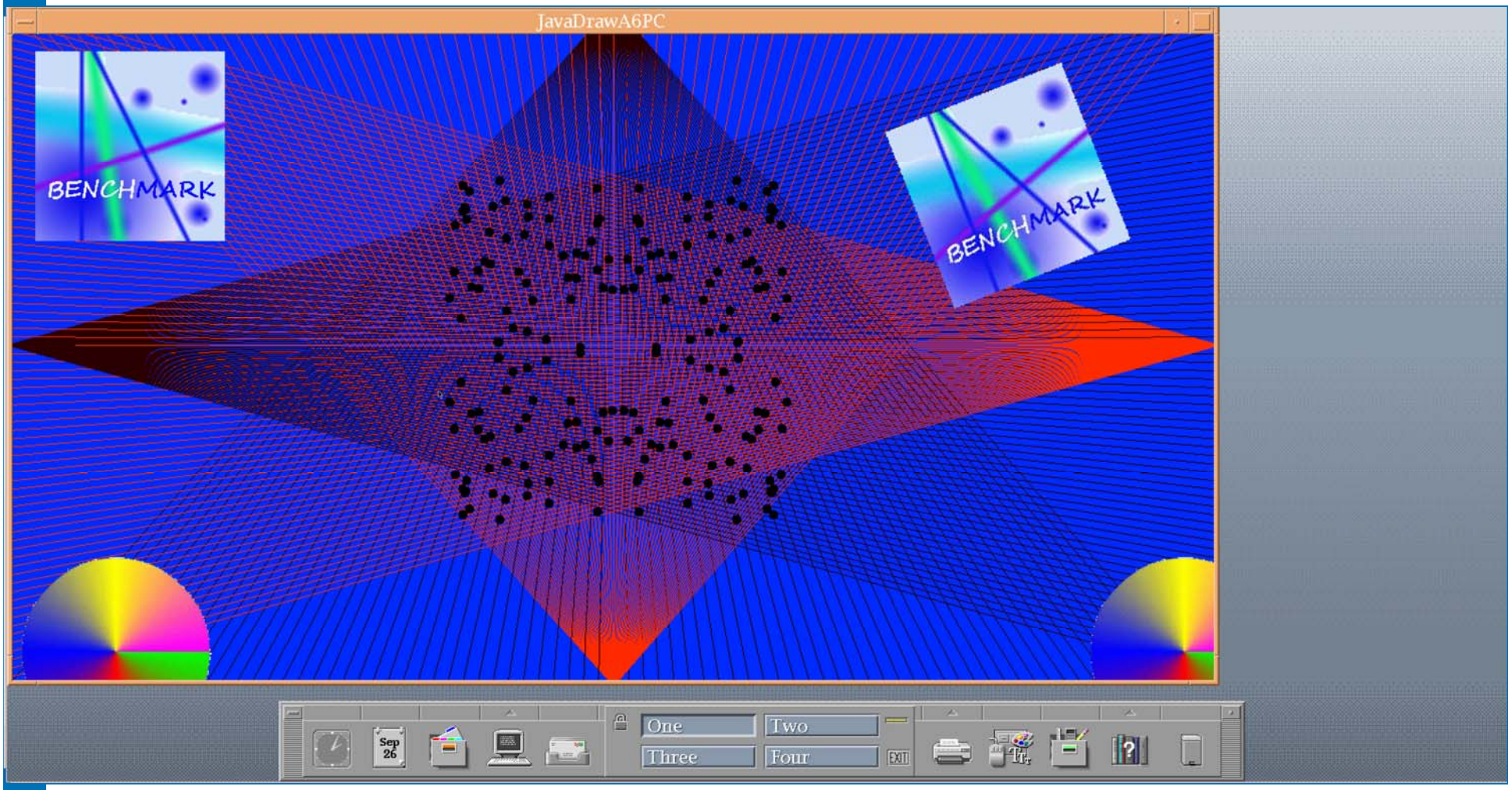
Java 8 Port : Current Status

- In the middle of round 5, debugging / round 6, filling in the blanks
- X-Windows / Motif integration

Most of it is working…



Most of it is working…



Java 7 new features

- Diamond Operator
  - Java 6:
```
Map<String, List<Trade>> trades = new
                    TreeMap<String, List<Trade>> ();
```
  - Java 7:
```
Map<String, List<Trade>> trades = new Treemap<> ();
```

- Strings in switch statement
  - Java 6: Series of if / else if statements
  - Java 7: switch statement with strings as case labels

- Numeric literals with underscores
  - Java 6:
```
int million = 1000000;
```
  - Java 7:
```
int million = 1_000_000;
```

Java 7 new features

- Automatic resource cleanup
  - Java 6:

```
try {
    FileOutputStream fos = new
FileOutputStream("movies.txt");
    DataOutputStream dos = new DataOutputStream(fos);
    dos.writeUTF("Java 7 Block Buster");
} catch(IOException e) {
    e.printStackTrace();
} finally {
    dos.close();
    fos.close();
}
```

  - Java 7:

```
try (FileOutputStream fos = new
FileOutputStream("movies.txt");
     DataOutputStream dos = new DataOutputStream(fos))
{
    dos.writeUTF("Java 7 Block Buster");
} catch(IOException e) {
    e.printStackTrace();
}
```

Java 7 new features

- Multi-exception catch
  - Java 6:

```
try {
    …
} catch(ExceptionTypeA e) {
    e.printStackTrace();
} catch(ExceptionTypeB e) {
    e.printStackTrace();
}
```

  - Java 7:

```
try {
    …
} catch(ExceptionTypeA | ExceptionTypeB e) {
    e.printStackTrace();
}
```

Java 7 new features

- New file system I/O (NIO 2.0)
  - More predictable across platforms
  - WatchService for file change notifications
- Easy “Fork and Join” paralellization
  - Using ForkJoinPool and ForkJoinTask
- Dynamic run-time function invocation
  - Not used by the Java language, but useful for dynamically-typed languages running on the JVM.

Java 8 new features

- Default method implementation in interface
  - Interface methods can now have a “default” implementation
  - If a class implements two interfaces that both provide a default implementation of the same method, the class has to override this method

Java 8 new features

- Functional interfaces and lambda expressions
  - Functional interface: interface with a single abstract method
  - Java 7:

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("My Runnable");
    }
};
```

  - Java 8:

```
Runnable r = () -> {
    System,out.println("My Runnable");
}
```

  - Or even:

```
Runnable r = () -> System.out.println("My Runnable");
```

Java 8 new features

- forEach function
  - Java 7:

```
Iterator<Integer> it = myList.iterator();
while (it.hasNext()) {
    Integer i = it.next();
    System.out.println(i);
}
```

  - Java 8:

```
myList.forEach(new Consumer<Integer>() {
    public void accept(Integer i) {
        System.out.println(i);
    }
});
```

  - Consumer is a functional interface, so with Lambda:

```
myList.forEach(i -> System.out.println(i));
```

Java 8 new features

- Stream API for Big Data
  - Parallel (or sequential) processing of collections

```
Stream<Integer> dataSet = myList.parallelStream();
Stream<Integer> underTen = myList.filter(i -> i<10);
underTen.forEach(i -> System.out.println(i));
```

- New Time API
  - Standardized approach to using time and date on Java

- Improvements in Collection, Concurrency, and IO API's

Credits

- Some Java 7 code examples taken from O'Reilly Publishing / Madhusudhan Konda