Incremental Improvements to the Java Platform

@PaulSandoz

NETFLIX

How do we know what “improved” between major Java releases?

How do we know what *changed* between major Java releases?

Releasing faster

	9 (2017/09)	10 (2018/03)	11 (2018/09)	12 (2019/03)
Targeted JEPs	jdk9	jdk/10	jdk/11	jdk/12
JSR	jdk9/spec/	jdk/10/spec	jdk/11/spec	jdk/12/spec
Fixed Issues	bugs.openjdk	bugs.openjdk	bugs.openjdk	bugs.openjdk
CSRs	N/A	bugs.openjdk	bugs.openjdk	bugs.openjdk
Release Notes	bugs.openjdk Oracle	bugs.openjdk Oracle	bugs.openjdk Oracle	bugs.openjdk

JEP = JDK Enhancement-Proposal JSR = Java Specification Request CSR = Compatibility & Specification Review

Build and look at the annotated source

```
# Get JDK 11
wget -qO- https://download.java.net/java/GA/jdk11/13/GPL/
openjdk-11.0.1_osx-x64_bin.tar.gz | tar xvz

# Clone the latest JDK
hg clone http://hg.openjdk.java.net/jdk/jdk/

# Configure with JDK 11 as the boot JDK
cd jdk
bash configure --with-boot-jdk=../jdk-11.0.1.jdk/
Contents/Home/

# Create the IDEA project
ANT_HOME=<...> bash bin/idea.sh

# Build the JDK
make images
```

I wish there was a way
in the published JDK JavaDoc to
filter all classes and methods
within a version range

What follows is
a set of improvements
unashamedly selected
by this presenter

Deduplicating lambdas
JDK-8200301
fixed in JDK 11
…
and a diversion into
a language feature

Comparison and mismatching over memory regions

Comparison and mismatching over arrays

- JDK 9 added support for array mismatch and lexicographical comparison; enhanced array equality
  - JDK-8033148 and related issues
- Mismatch: find the relative index of the first mismatching pair of elements in two arrays
  - Comparison and equality are composed from mismatch

```
compareUnsigned(byte[], byte[])
compareUnsigned(byte[], int, int, byte[], int, int)
```

```
compareUnsigned(byte[], byte[])
compareUnsigned(byte[], int, int, byte[], int, int)
compareUnsigned(int[], int[])
compareUnsigned(int[], int, int, int[], int, int)
compareUnsigned(long[], long[])
compareUnsigned(long[], int, int, long[], int, int)
compareUnsigned(short[], short[])
compareUnsigned(short[], int, int, short[], int, int)

compare(boolean[], boolean[])
compare(boolean[], int, int, boolean[], int, int)
compare(byte[], byte[])
compare(byte[], int, int, byte[], int, int)
compare(char[], char[])
compare(char[], int, int, char[], int, int)
compare(double[], double[])
compare(double[], int, int, double[], int, int)
compare(float[], float[])
compare(float[], int, int, float[], int, int)
compare(int[], int[])
compare(int[], int, int, int[], int, int)
compare(long[], long[])
compare(long[], int, int, long[], int, int)
compare(short[], short[])
compare(short[], int, int, short[], int, int)
compare(T[], T[])
compare(T[], int, int, T[], int, int)
compare(T[], T[], Comparator)
compare(T[], int, int, T[], int, int, Comparator)
```

```
equals(boolean[], int, int, boolean[], int, int)
equals(byte[], int, int, byte[], int, int)
equals(char[], int, int, char[], int, int)
equals(double[], int, int, double[], int, int)
equals(float[], int, int, float[], int, int)
equals(int[], int, int, int[], int, int)
equals(long[], int, int, long[], int, int)
equals(short[], int, int, short[], int, int)
equals(Object[], int, int, Object[], int, int)

mismatch(boolean[], boolean[])
mismatch(boolean[], int, int, boolean[], int, int)
mismatch(byte[], byte[])
mismatch(byte[], int, int, byte[], int, int)
mismatch(char[], char[])
mismatch(char[], int, int, char[], int, int)
mismatch(double[], double[])
mismatch(double[], int, int, double[], int, int)
mismatch(float[], float[])
mismatch(float[], int, int, float[], int, int)
mismatch(int[], int[])
mismatch(int[], int, int, int[], int, int)
mismatch(long[], long[])
mismatch(long[], int, int, long[], int, int)
mismatch(short[], short[])
mismatch(short[], int, int, short[], int, int)
mismatch(Object[], Object[])
mismatch(Object[], int, int, Object[], int, int)
mismatch(T[], T[], Comparator)
```

Comparison and mismatching over arrays

- The majority of the implementation effort was spent supporting a highly performant implementation for primitive arrays
  - And there are always edge cases to manage e.g. double/ float NaN
- Leverages vectorized mismatch intrinsic in Hotspot
  - x86 implementation (contributed by Intel) uses SIMD (AVX) instructions

```
java -XX:+UnlockDiagnosticVMOptions \
  -XX:+PrintStubCode -version \
  | grep StubRoutines::vectorizedMismatch
```

Comparison and mismatching over nio Buffers

- JDK 11 enhanced the performance of nio Buffer equality and comparison (JDK-8193085) building on prior work in JDK 9
  - Vectorized mismatch intrinsic used in arrays
  - Unifying on and off-heap access required for atomic access to Buffers (see VarHandles)
- JDK 11 added support for Buffer mismatch (JDK-8202216)

Executing Java source

Executing Java source

- JDK 11 added support for executing Java source without explicit compilation
  - JEP 330
  - No change to the Java Language Specification
- Lowering Java's "activation energy"

Executing Java source

- Can use with #!
- Further incremental improvements may be possible
  - No class declaration and main method (jshell like)?

Collection literals

Collection literals

- Introduced “small” library feature in JDK 9 instead of a language change
  - **`List.of`**, **`Set.of`**, **`Map.of`**
    - Unmodifable and **`null`** hostile
- **`List`**/**`Set`**/**`Map.copyOf`** methods added in JDK 10
- Incremental performance improvements in JDK 11 (JDK-8193128)

`Optional.orElseThrow`
in JDK 10 (JDK-8193280)

Use **`orElseThrow`** instead of **`Optional.get`**

`Predicate.not`

in JDK 11 (JDK-8203428)

```
stream
.filter(not(String::isEmpty))
```

`Collection.toArray(IntFunction<T[]> )`

in JDK 11 (JDK-8193155)

Just like **`Stream.toArray`**

```
String[] a = c.toArray(String[]::new)
```

Methods on InputStream

Methods on InputStream

- JDK 9
  - **`byte[] readAllBytes()`**
  - **`int readNBytes(byte[] b, int off, int len)`**
  - **`long transferTo(OutputStream out)`**
- JDK 11
  - **`byte[] readNBytes(int len)`**
    (**`readAllBytes`** was updated to call **`readNBytes(Integer.MAX_VALUE)`**)
  - **`InputStream nullInputStream()`**
    (and companion on OutputStream)

Methods on String

Methods on String

- JDK 11
  - **`strip()`**, **`stripLeading()`**, **`stripTrailing()`** (Removes whitespace, **`trim`** removes <= U+0020)
  - **`isBlank()`**
  - **`Stream<String> lines()`**
  - **`repeat(int count)`**

Methods on String

- JDK 12
  - **`indent(int n)`**
  - **`align()`**
  - **`align(int n)`**
- Incremental improvements to complement raw string literals
  - Previewed in JEP 326

Files and Strings

- Methods added in JDK 11 to Files
  - **`String readString(Path path)`**
  - **`Path writeString(Path path, CharSequence csq, OpenOption... options)`**
  - And Charset accepting variants

HotSpot enhancements

HotSpot enhancements

- Fused Multiply and Add (FMA) vectorization on x86
  - JDK-8181616, in JDK 10
  - One rounding error for **`Math.fma(a, b, c)`** instead of two with **`a * b + c`**

Performance

Benchmark		(size)	Mode	Cnt	Score	Error	Units
DotProduct.fma	8	8096	avgt	5	8400.144 ±	674.883	ns/op
DotProduct.fmaUnrolled	8	8096	avgt	5	2292.339 ±	163.906	ns/op
DotProduct.fmaVectorized	8	8096	avgt	5	4384.019 ±	157.347	ns/op
DotProduct.fmaVectorizedTheoretical	8	8096	avgt	5	1042.215 ±	17.552	ns/op
DotProduct.scalar	8	8096	avgt	5	8476.846 ±	358.579	ns/op
DotProduct.vectorApi	8	8096	avgt	5	1012.717 ±	39.083	ns/op
DotProduct.vectorApiUnrolled	8	8096	avgt	5	335.321 ±	3.472	ns/op

No data parallelism for scalar and explicit fma,
as expected due to Java's requirements on reproducible floating point calculations

Performance

```
Benchmark                                    (size)  Mode  Cnt     Score     Error  Units
DotProduct.fma                                 8096  avgt    5  8400.144 ± 674.883  ns/op
DotProduct.fmaUnrolled                         8096  avgt    5  2292.339 ± 163.906  ns/op
DotProduct.fmaVectorized                       8096  avgt    5  4384.019 ± 157.347  ns/op
DotProduct.fmaVectorizedTheoretical            8096  avgt    5  1042.215 ±  17.552  ns/op
DotProduct.scalar                              8096  avgt    5  8476.846 ± 358.579  ns/op
DotProduct.vectorApi                           8096  avgt    5  1012.717 ±  39.083  ns/op
DotProduct.vectorApiUnrolled                   8096  avgt    5   335.321 ±   3.472  ns/op
```

When the **`vfmadd231ps`** instruction kicks in (in this case using the Vector API) performance gets much better, especially when the data dependency is overcome

HotSpot enhancements

- Square root vectorization on **`float`**s on x86
  - JDK-8190800, fixed in JDK 10
  - Previously only **`double`** was supported
- Many more, see “Up and Away: JDK Optimizations Beyond JDK 9” [DEV5091]

Performance

```
java -XX:-TieredCompilation -XX:-UseSuperWord -jar target/benchmarks.jar -prof
dtraceasm Sqrt

Benchmark                (size)  Mode  Cnt      Score    Error  Units
Sqrt.sqrt_float            8096  avgt    5  13545.204 ± 71.103  ns/op

                ↗  0x000000010bc57f50: vsqrtss 0x10(%r13,%r11,4),%xmm1,%xmm1
  4.68%         │  0x000000010bc57f57: vmovss %xmm1,0x10(%rax,%r11,4)
  0.29%         │  0x000000010bc57f5e: vsqrtss 0x14(%r13,%r11,4),%xmm0,%xmm0
 40.92%         │  0x000000010bc57f65: vmovss %xmm0,0x14(%rax,%r11,4)
  4.03%         │  0x000000010bc57f6c: vsqrtss 0x18(%r13,%r11,4),%xmm1,%xmm1
  1.15%         │  0x000000010bc57f73: vmovss %xmm1,0x18(%rax,%r11,4)
  0.18%         │  0x000000010bc57f7a: vsqrtss 0x1c(%r13,%r11,4),%xmm0,%xmm0
 44.27%         │  0x000000010bc57f81: vmovss %xmm0,0x1c(%rax,%r11,4)
  4.21%         │  0x000000010bc57f88: add    $0x4,%r11d
  0.02%         │  0x000000010bc57f8c: cmp    %r8d,%r11d
                ↘  0x000000010bc57f8f: jl     0x000000010bc57f50
```

Performance
~8x better

```
java -XX:-TieredCompilation -XX:+UseSuperWord -jar target/benchmarks.jar -prof
dtraceasm Sqrt

Benchmark              (size)  Mode  Cnt     Score    Error  Units
Sqrt.sqrt_float          8096  avgt    5  1717.071 ± 23.885  ns/op

 0.02%    ↗↗  0x000000010e820460: vsqrtps 0x10(%rsi,%r11,4),%ymm0
19.90%    ||  0x000000010e820467: vmovdqu %ymm0,0x10(%rax,%r11,4)
 4.23%    ||  0x000000010e82046e: movslq %r11d,%rbx
          ||  0x000000010e820471: vsqrtps 0x30(%rsi,%rbx,4),%ymm0
20.81%    ||  0x000000010e820477: vmovdqu %ymm0,0x30(%rax,%rbx,4)
 4.31%    ||  0x000000010e82047d: vsqrtps 0x50(%rsi,%rbx,4),%ymm0
20.52%    ||  0x000000010e820483: vmovdqu %ymm0,0x50(%rax,%rbx,4)
 3.66%    ||  0x000000010e820489: vsqrtps 0x70(%rsi,%rbx,4),%ymm0
20.52%    ||  0x000000010e82048f: vmovdqu %ymm0,0x70(%rax,%rbx,4)
 4.00%    ||  0x000000010e820495: add    $0x20,%r11d
          ||  0x000000010e820499: cmp    %r9d,%r11d
          ||  0x000000010e82049c: jl     0x000000010e820460
```

?