Demystifying the most significant Java language features from 9 to 11

by Ionuţ Baloşin

IonutBalosin.com

@IonutBalosin

About Me

Software Architect Raiffeisen Bank International

Technical Trainer

Java Performance Tuning

Software Architecture Essentials

Designing High-Performance Applications

< for more details please visit: IonutBalosin.com/training >

Writer IonutBalosin.com InfoQ

Speaker

DEVOX™ VOXXED SHARE THE KNOWLEDGE Joker<?> Dev<Talks/> DevDays JBCNConf BARCELONA JAVA USERS GROUP

ORACLE CODE geecon Riga DevDays jPrime I.T.A.K.E. Unconference DevFest Vienna XPDAYS BENELUX geekOUT {"ON": "THE BEACH"}

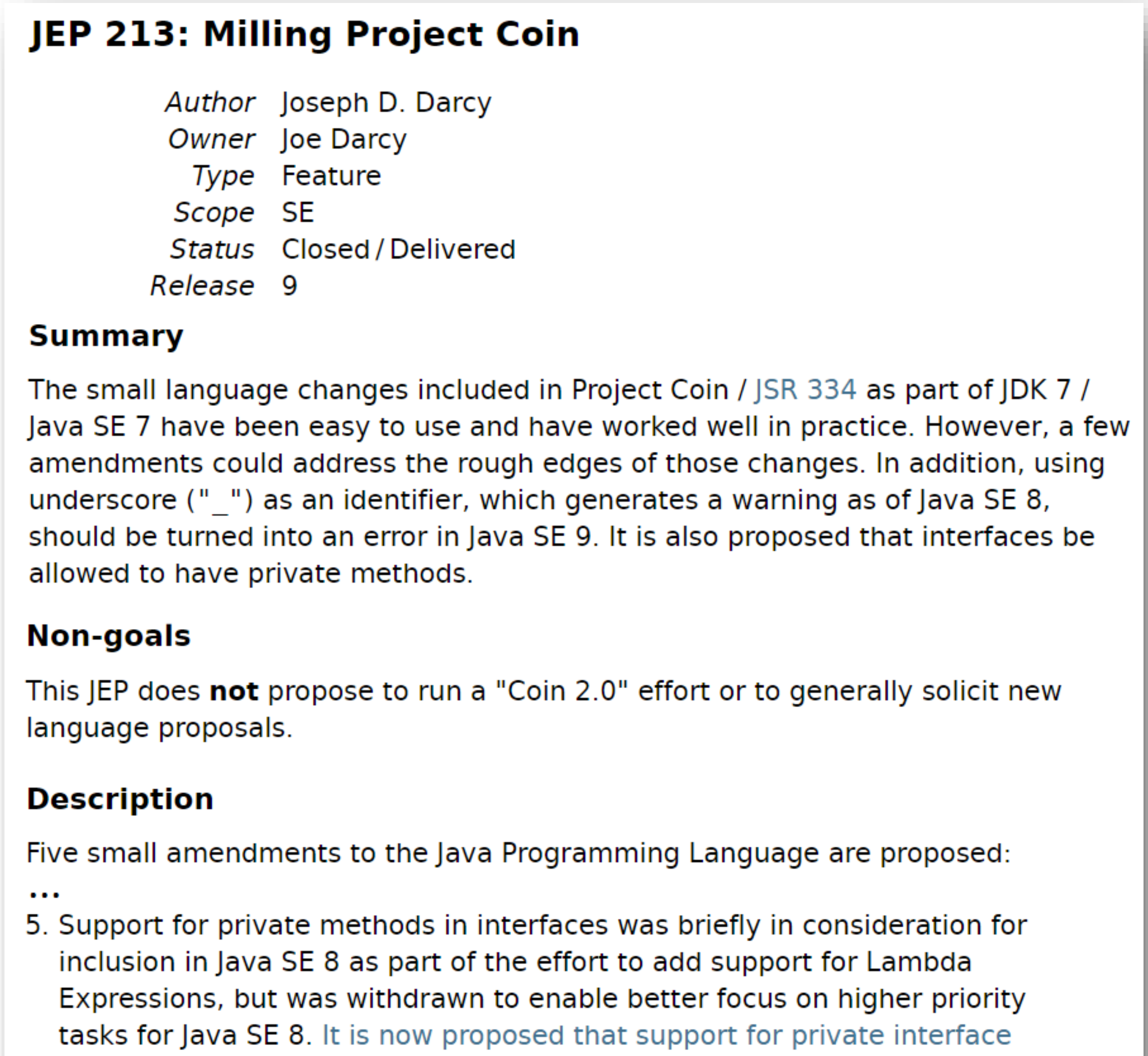
Today’s Content

01 Private Interface Methods
02 Compact Strings
03 Indify String Concatenation
04 Stack-Walking API
05 Spin-Wait Hints
06 Collections Factory Methods
07 Local-Variable Type Inference
08 Nest-Based Access Control

September 2017	**JDK 9**	JEP 213 **Private Interface Methods**
		JEP 254 **Compact Strings**
		JEP 280 **Indify String Concatenation**
		JEP 259 **Stack-Walking API**
		JEP 285 **Spin-Wait Hints**
		JEP 269 **Collections Factory Methods**
March 2018	**JDK 10**	JEP 286 **Local-Variable Type Inference**
September 2018	**JDK 11**	JEP 181 **Nest-Based Access Control**

Private Interface Methods

01


JEP 213: Milling Project Coin

Author	Joseph D. Darcy
Owner	Joe Darcy
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9

Summary

The small language changes included in Project Coin / JSR 334 as part of JDK 7 / Java SE 7 have been easy to use and have worked well in practice. However, a few amendments could address the rough edges of those changes. In addition, using underscore ("_") as an identifier, which generates a warning as of Java SE 8, should be turned into an error in Java SE 9. It is also proposed that interfaces be allowed to have private methods.

Non-goals

This JEP does **not** propose to run a "Coin 2.0" effort or to generally solicit new language proposals.

Description

Five small amendments to the Java Programming Language are proposed:

...

5. Support for private methods in interfaces was briefly in consideration for inclusion in Java SE 8 as part of the effort to add support for Lambda Expressions, but was withdrawn to enable better focus on higher priority tasks for Java SE 8. It is now proposed that support for private interface

```
public interface IPrivateMethod {
    default void foo() {
        print();
    }
    private void print() {
        System.out.println("Private Method");
    }
}
```

private interface method

```
public interface IPrivateMethod {
    default void foo() {
        print();
    }
    private void print() {
        System.out.println("Private Method");
    }
}
```

private interface method

BYTECODE

```
public void foo();
    aload_0
    invokeinterface #1, 1 // InterfaceMethod print:()V

private void print();
    getstatic     #2      // Field System.out:LPrintStream;
    ldc           #3      // String "Private Method"
    invokevirtual #4      // Method PrintStream.println:(LString;)V
```

private interface method is called using INVOKEINTERFACE, as any other interface method.

default is a keyword present only at the language level, it doesn't exist at bytecode level.

```java
public interface IPrivateMethod {
    static void baz() {
        staticPrint();
    }
    private static void staticPrint() {
        System.out.println("Private Static Method");
    }
}
```

static private interface method

```
public interface IPrivateMethod {
    static void baz() {
        staticPrint();
    }
    private static void staticPrint() {
        System.out.println("Private Static Method");
    }
}
```

static private interface method

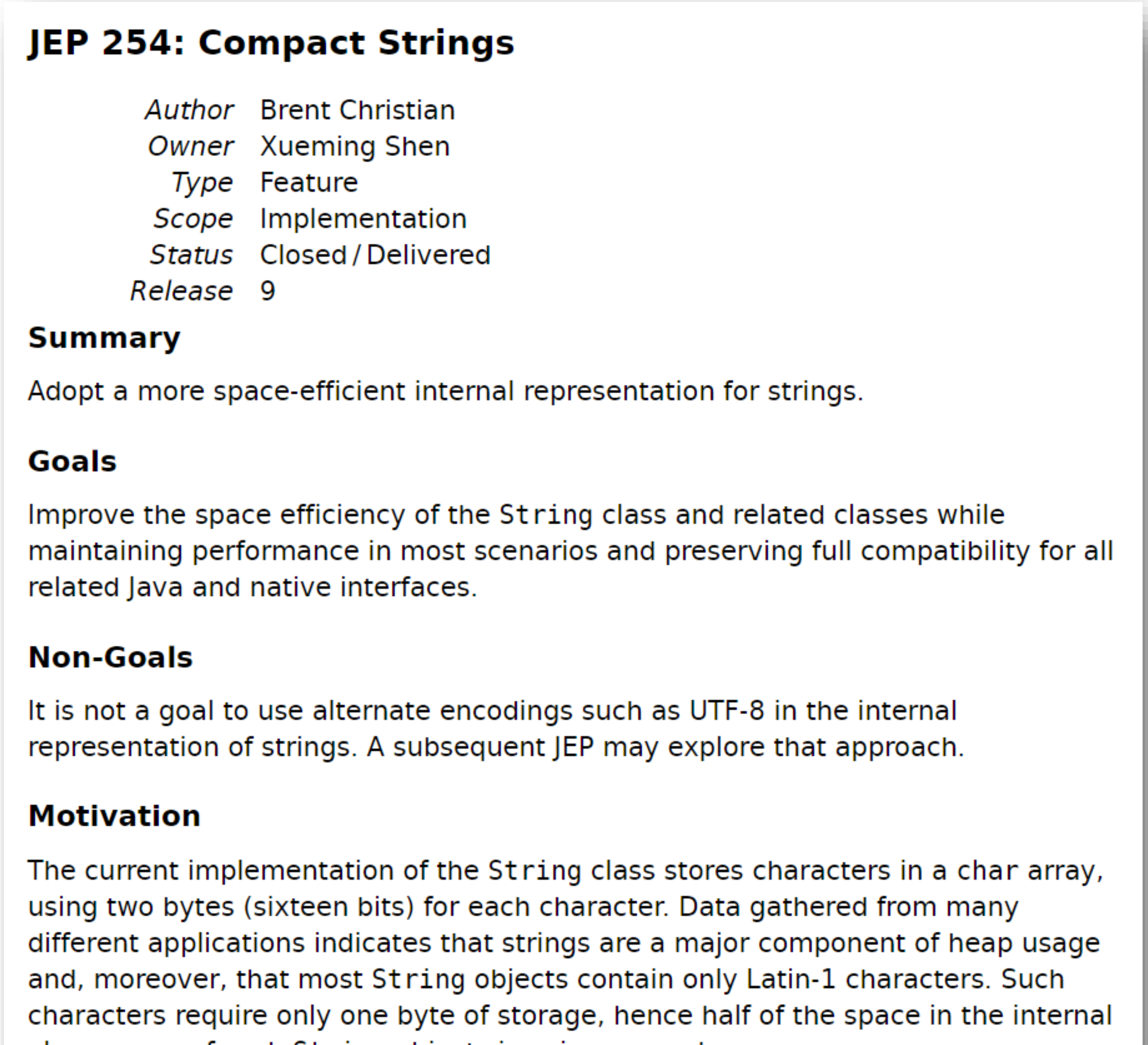
BYTECODE

```
public static void baz();
    invokestatic  #5    // InterfaceMethod staticPrint:()V

private static void staticPrint();
    getstatic     #2    // Field System.out:LPrintStream;
    ldc           #6    // String "Private Static Method"
    invokevirtual #4    // Method PrintStream.println:(LString;)V
```

private static interface method is called using INVOKESTATIC, as any static class method.

Compact Strings

02



JEP 254: Compact Strings

Author	Brent Christian
Owner	Xueming Shen
Type	Feature
Scope	Implementation
Status	Closed / Delivered
Release	9

Summary

Adopt a more space-efficient internal representation for strings.

Goals

Improve the space efficiency of the `String` class and related classes while maintaining performance in most scenarios and preserving full compatibility for all related Java and native interfaces.

Non-Goals

It is not a goal to use alternate encodings such as UTF-8 in the internal representation of strings. A subsequent JEP may explore that approach.

Motivation

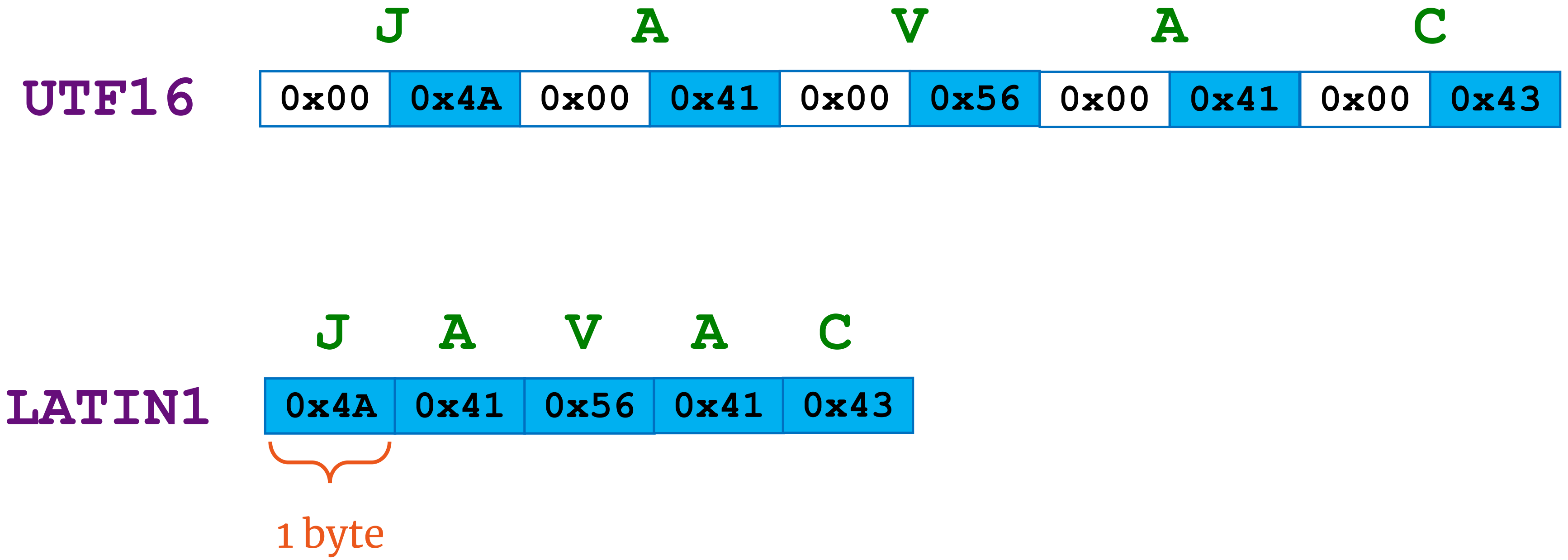
The current implementation of the `String` class stores characters in a `char` array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most `String` objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal

```
public final class String {

    @Stable
    private final byte[] value;  <- JDK 8: private final char value[];
    private int hash;

    private final byte coder;  // LATIN1 || UTF16

    @Native static final byte LATIN1 = 0;
    @Native static final byte UTF16  = 1;

    static final boolean COMPACT_STRINGS;
    static {
        COMPACT_STRINGS = true;
    }
}
```

```
public final class String {

    @Stable
    private final byte[] value; <- JDK 8: private final char value[];
    private int hash;

    private final byte coder;  // LATIN1 || UTF16

    @Native static final byte LATIN1 = 0;
    @Native static final byte UTF16  = 1;

    static final boolean COMPACT_STRINGS;
        static {
            COMPACT_STRINGS = true;
        }
}
```

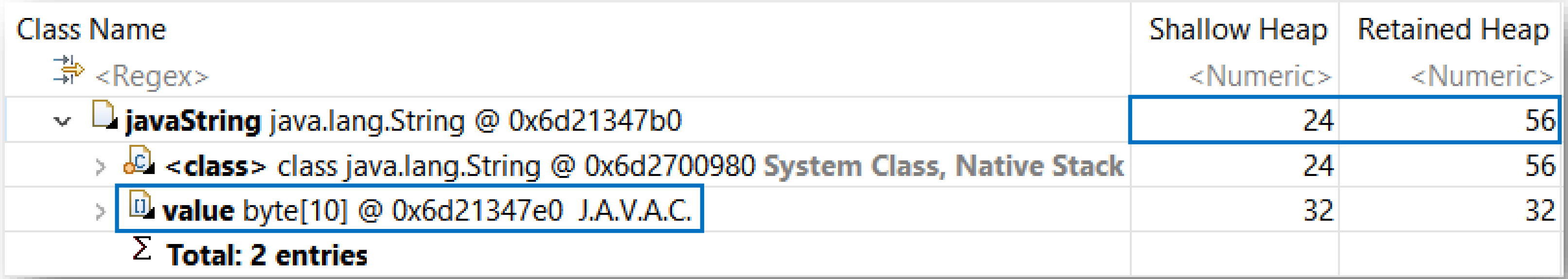
Note: -XX:+CompactStrings is enabled by default

```
String javaString = new String("JAVAC");
```



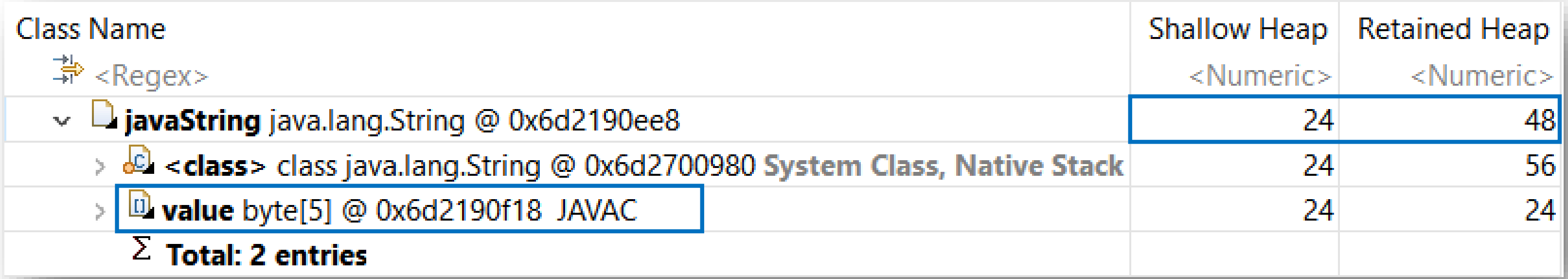
```java
String javaString = new String("JAVAC");
```

UTF16



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
javaString java.lang.String @ 0x6d21347b0	24	56
<class> class java.lang.String @ 0x6d2700980 **System Class, Native Stack**	24	56
value byte[10] @ 0x6d21347e0 J.A.V.A.C.	32	32
Σ **Total: 2 entries**		

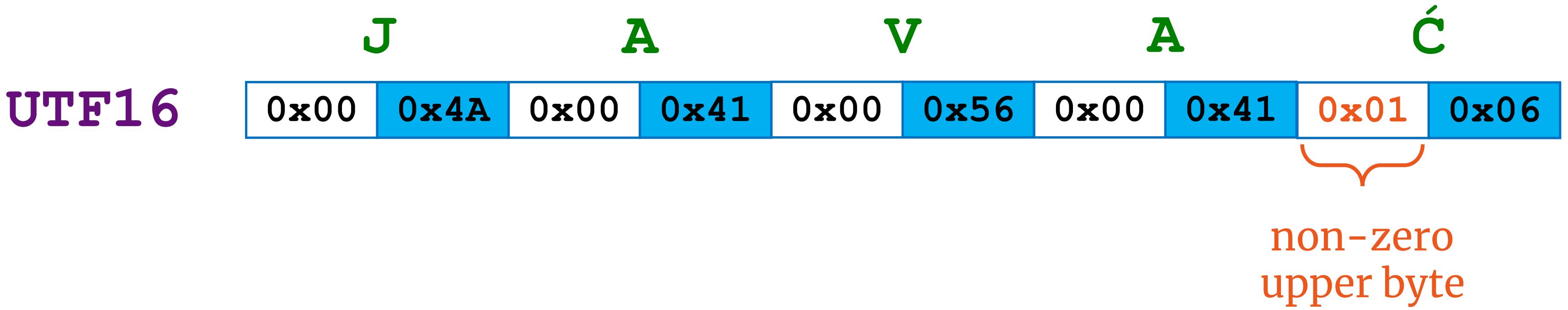
LATIN1



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
javaString java.lang.String @ 0x6d2190ee8	24	48
<class> class java.lang.String @ 0x6d2700980 **System Class, Native Stack**	24	56
value byte[5] @ 0x6d2190f18 JAVAC	24	24
Σ **Total: 2 entries**		

UTF16

```java
String javaString = new String("JAVAĆ");
```



Strings containing characters with non-zero upper byte cannot be compressed. They are stored as 2 byte characters using UTF16 encoding.

String Construction

```
// @see java.lang.String.java
String(char[] value, int off, int len, Void sig) {
    // ...
    if (COMPACT_STRINGS) {
        byte[] val = StringUTF16.compress(value, off, len);
        if (val != null) {
            this.value = val;
            this.coder = LATIN1;
            return;
        }
    }
    this.coder = UTF16;
    this.value = StringUTF16.toBytes(value, off, len);
}
```

1. first, it tries to compress the input char[] to Latin1 by stripping off upper bytes
2. if it fails, UTF16 encoding is used (i.e. each char spreads across 2 bytes)

```java
@Param({"ÐžÐ¹,Ð²ÑÑ‘Ð"})
public String utf_16_str1;

@Param({"ϑ¿Ñ€Ð¾Ð¿φÐ°X"})
public String utf_16_str2;

@Param({"ÐϛϚ»Ð¾,ÑˆÐµÑ„"})
public String utf_16_str3;
```

```java
@Param({"ΦÈ¾ʬXÐ»ÐΛϼȢË"})
public String utf_16_str4;

@Param({"'ΩΔΘΞΨθϛώϚϡȢϱ"})
public String utf_16_str5;

@Param({"XÐϚϡΨθϑ¿ÐžÈ¾"})
public String utf_16_str6;
```

```java
@Benchmark
public String utf16_multiple_concat() {
    return utf_16_str1 + utf_16_str2 + utf_16_str3 +
           utf_16_str4 + utf_16_str5 + utf_16_str6;
}
```

default

	Average Time (ns/op)	Allocation Rate (B/op)
-XX:+CompactStrings	54.234 ± 2.081	192.0
-XX:-CompactStrings	45.957 ± 1.777	192.0

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

default

	Average Time (ns/op)	Allocation Rate (B/op)
-XX:+CompactStrings	54.234 ± 2.081	192.0
-XX:-CompactStrings	45.957 ± 1.777	192.0

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

For applications that extensively use **UTF16** characters, consider disabling **Compact Strings** for a better performance.

String.equals()

```java
// @see java.lang.String.java
public boolean equals(Object anObject) {
    // ...
    if (anObject instanceof String) {
        String aString = (String)anObject;
        if (coder() == aString.coder()) {                    1
            return isLatin1() ? StringLatin1.equals(value, aString.value)
                              : StringUTF16.equals(value, aString.value);
        }
    }
    return false;       2
}
```

```
@Param({"123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"}) // length 35
public String latin_1;

@Param({"123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"})
public String latin_1_identical;
```

UTF16

```
@Param({"123456789ABCDEFGHIJKLMNOPQRSTUVWX¾ʬ"}) // length 35
public String utf_16;

@Param({"123456789ABCDEFGHIJKLMNOPQRSTUVWX¾ʬ"})
public String utf_16_identical;
```

```java
@Benchmark
public boolean latin1_toLatin1_equals() {
    return latin_1.equals(latin_1_identical);
}

@Benchmark
public boolean latin1_toUtf16_equals() {
    return latin_1.equals(utf_16);
}

@Benchmark
public boolean utf16_toUtf16_equals() {
    return utf_16.equals(utf_16_identical);
}
```

default

Average Time (ns/op)

	-XX:+CompactStrings	-XX:-CompactStrings
latin1_toLatin1_equals	6.745 ± 0.418	7.793 ± 0.411
utf16_toUtf16_equals	7.910 ± 0.624	7.491 ± 0.372
latin1_toUtf16_equals	3.316 ± 0.201	7.114 ± 0.301

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

default

Average Time (ns/op)

	-XX:+CompactStrings	-XX:-CompactStrings
latin1_toLatin1_equals	6.745 ± 0.418	7.793 ± 0.411
utf16_toUtf16_equals	7.910 ± 0.624	7.491 ± 0.372
latin1_toUtf16_equals	3.316 ± 0.201	7.114 ± 0.301

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

-XX:+CompactStrings:
- if coders differ, Strings comparison is really cheap.
- if the same coder, slightly better performance for Latin1.

-XX:-CompactStrings:
- almost the same response time in all cases.

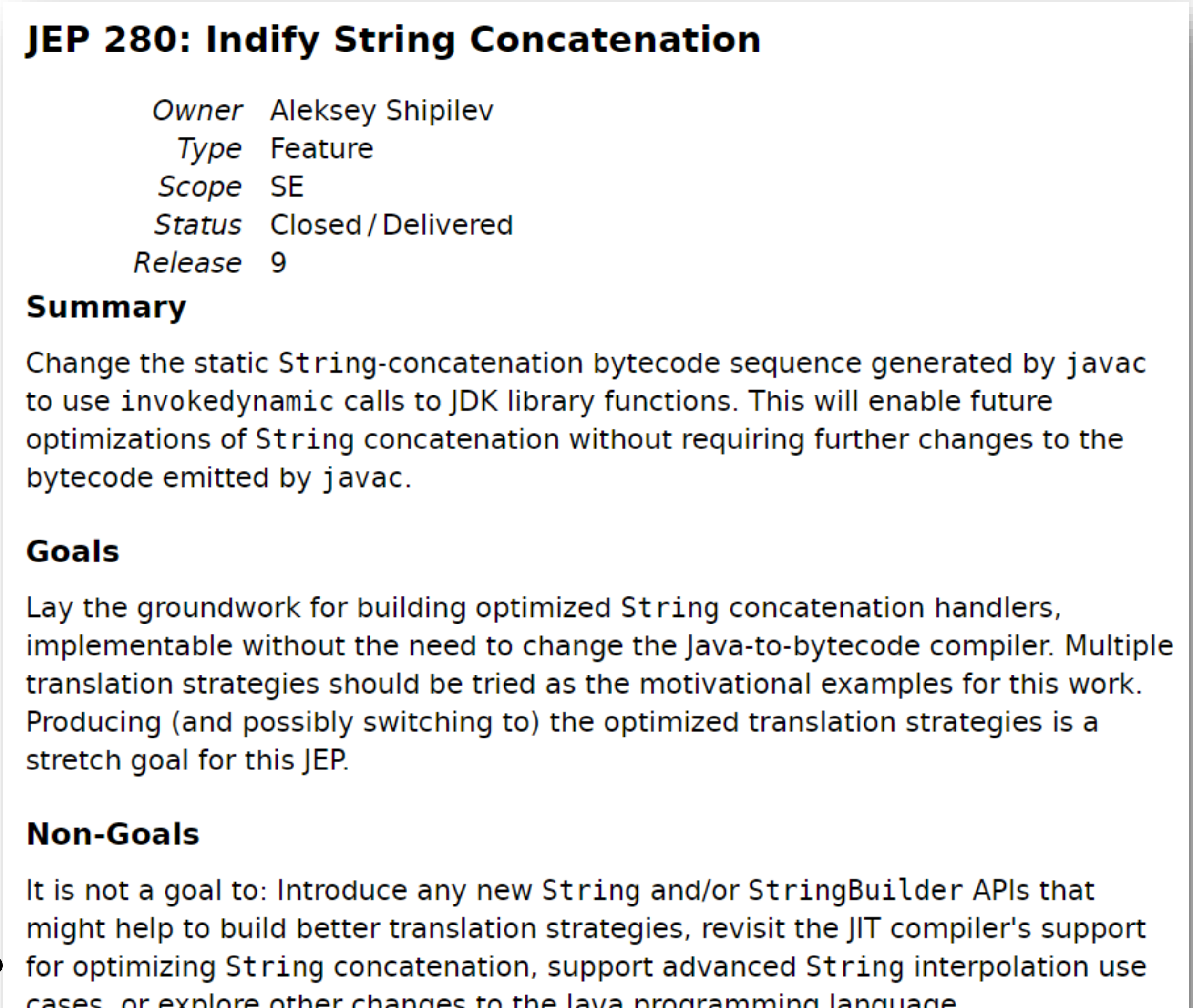
Indify String Concatenation

03

```
private String aString;    // aString::length = 128
private int anInt;         // anInt = 2019
private float aFloat;      // aFloat = 42.0f

@Benchmark
public String concat() {
    return aString + anInt + aFloat;
}
```

JAVA 8 GENERATED BYTECODE

```
public String concat();
    new             #2  // class StringBuilder
    dup
    invokespecial #12 // Method StringBuilder."<init>":()V
    aload_0
    getfield        #8  // Field aString:LString;
    invokevirtual #6  // Method StringBuilder.append:(LString;)LStringBuilder;
    aload_0
    getfield        #9  // Field anInt:I
    invokevirtual #13 // Method StringBuilder.append:(I)LStringBuilder;
    aload_0
    getfield        #11 // Field aFloat:F
    invokevirtual #14 // Method StringBuilder.append:(F)LStringBuilder;
    invokevirtual #7  // Method StringBuilder.toString:()LString;
    areturn
```



JEP 280: Indify String Concatenation

Owner	Aleksey Shipilev
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9

Summary

Change the static `String`-concatenation bytecode sequence generated by `javac` to use `invokedynamic` calls to JDK library functions. This will enable future optimizations of `String` concatenation without requiring further changes to the bytecode emitted by `javac`.

Goals

Lay the groundwork for building optimized `String` concatenation handlers, implementable without the need to change the Java-to-bytecode compiler. Multiple translation strategies should be tried as the motivational examples for this work. Producing (and possibly switching to) the optimized translation strategies is a stretch goal for this JEP.

Non-Goals

It is not a goal to: Introduce any new `String` and/or `StringBuilder` APIs that might help to build better translation strategies, revisit the JIT compiler's support for optimizing `String` concatenation, support advanced `String` interpolation use cases, or explore other changes to the Java programming language

JAVA 11 GENERATED BYTECODE

```
public String concat();
     aload_0
     getfield      #8        // Field aString:Ljava/lang/String;
     aload_0
     getfield      #9        // Field anInt:I
     aload_0
     getfield      #11       // Field aFloat:F
     invokedynamic #12,  0   // InvokeDynamic #0:makeConcatWithConstants
     areturn

BootstrapMethods:
     REF_invokeStatic StringConcatFactory.makeConcatWithConstants:()LCallSite;
     Method arguments:
         \u0001\u0001\u0001
```

String Concat Strategies

```
-Djava.lang.invoke.stringConcat=<strategy>
```

Bytecode Generator

BC_SB - spins up the bytecode that has the same StringBuilder chain javac would emit. It moves the javac generated bytecode to runtime. **Baseline for current JDK behavior**.

BC_SB_SIZED - similar to **BC_SB**, but it tries to guess the capacity required for StringBuilder to accept all arguments without resizing. It only guesses the space required for known types (e.g. primitives and Strings), but does not convert arguments. The capacity estimate may be wrong and StringBuilder may have to resize.

BC_SB_SIZED_EXACT - improves over **BC_SB_SIZED**, by first converting all arguments to String in order to get the exact capacity StringBuilder should have.

Bytecode Generator

StringBuilder

Method Handle

BC_SB - spins up the bytecode that has the same StringBuilder chain javac would emit. It moves the javac generated bytecode to runtime. **Baseline for current JDK behavior**.

BC_SB_SIZED - similar to **BC_SB**, but it tries to guess the capacity required for StringBuilder to accept all arguments without resizing. It only guesses the space required for known types (e.g. primitives and Strings), but does not convert arguments. The capacity estimate may be wrong and StringBuilder may have to resize.

BC_SB_SIZED_EXACT - improves over **BC_SB_SIZED**, by first converting all arguments to String in order to get the exact capacity StringBuilder should have.

MH_SB_SIZED - builds the computation on public MethodHandle APIs which, at the end, ends up in calling the public StringBuilder API. This strategy does not use any private API at all.

MH_SB_SIZED_EXACT – improved over **MH_SB_SIZED** by first trying to get the exact capacity StringBuilder should have.

MH_INLINE_SIZED_EXACT - builds the char[] array on its own and passes that char[] array to String constructor. It opens the door for building a very optimal concatenation sequence, it does not use StringBuilder. **Default strategy**.

	Average Time (ns/op)	Allocation Rate (B/op)
JRE 1.8.0_211	134.996 ± 3.133	1,192.0
BC_SB	120.501 ± 5.621	656.0
BC_SB_SIZED	82.234 ± 3.297	384.0
BC_SB_SIZED_EXACT	59.217 ± 2.818	224.0
MH_SB_SIZED	88.952 ± 3.799	384.0
MH_SB_SIZED_EXACT	87.715 ± 3.124	376.0
MH_INLINE_SIZED_EXACT	54.773 ± 2.260	200.0

JRE 11.0.2

	Average Time (ns/op)	Allocation Rate (B/op)
JRE 1.8.0_211	134.996 ± 3.133	1,192.0
BC_SB	120.501 ± 5.621	656.0
BC_SB_SIZED	82.234 ± 3.297	384.0
BC_SB_SIZED_EXACT	59.217 ± 2.818	224.0
MH_SB_SIZED	88.952 ± 3.799	384.0
MH_SB_SIZED_EXACT	87.715 ± 3.124	376.0
MH_INLINE_SIZED_EXACT	54.773 ± 2.260	200.0

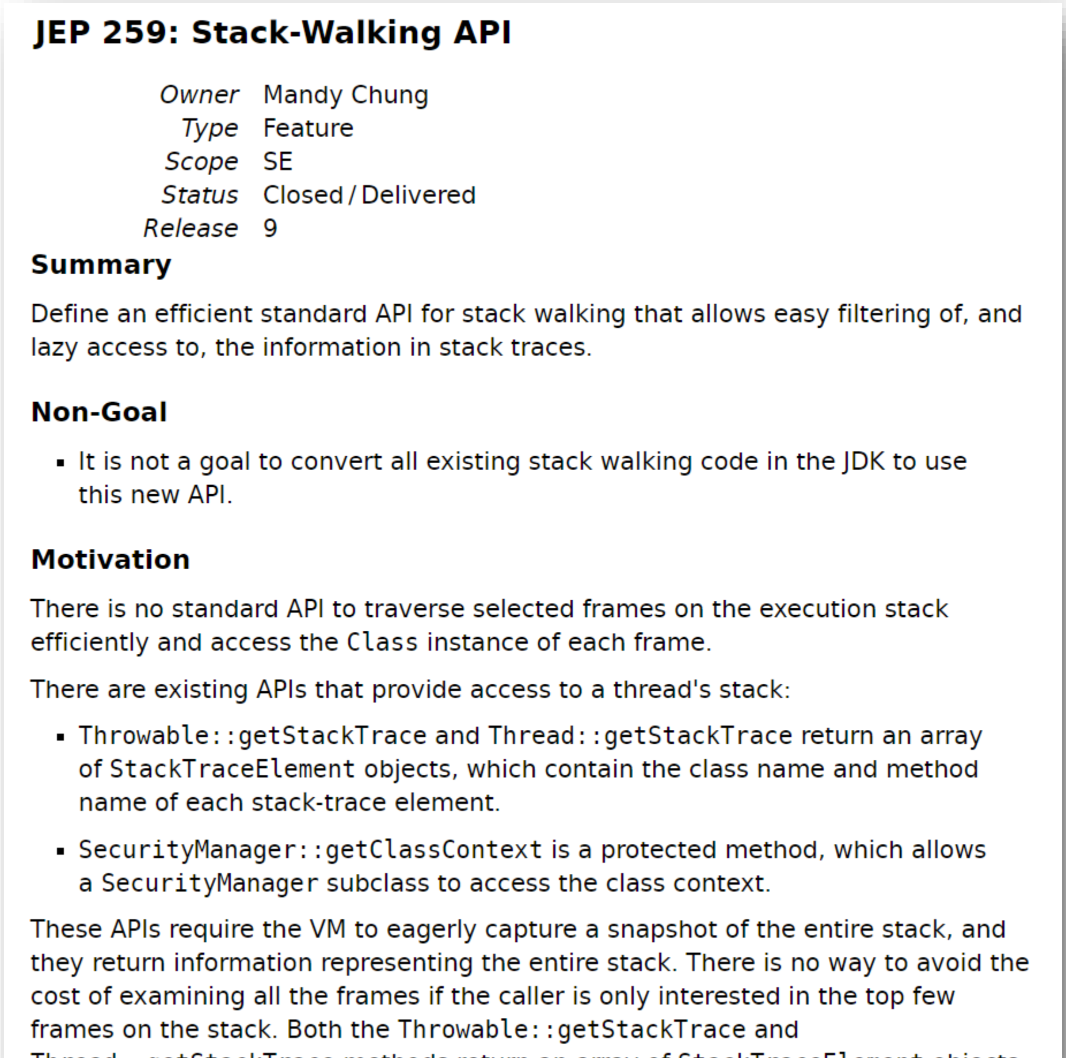
JRE 11.0.2

Advanced strategies (e.g. **MH_INLINE_SIZED_EXACT**) can improve performance when JIT compiler is not able to optimize the append StringBuilder chains (e.g. ***_SB_*** strategies).

Such advanced strategies can bring noticeable performance and allocation pressure improvements.

Stack-Walking API

04



JEP 259: Stack-Walking API

Owner	Mandy Chung
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9

Summary

Define an efficient standard API for stack walking that allows easy filtering of, and lazy access to, the information in stack traces.

Non-Goal

- It is not a goal to convert all existing stack walking code in the JDK to use this new API.

Motivation

There is no standard API to traverse selected frames on the execution stack efficiently and access the `Class` instance of each frame.

There are existing APIs that provide access to a thread's stack:

- `Throwable::getStackTrace` and `Thread::getStackTrace` return an array of `StackTraceElement` objects, which contain the class name and method name of each stack-trace element.
- `SecurityManager::getClassContext` is a protected method, which allows a `SecurityManager` subclass to access the class context.

These APIs require the VM to eagerly capture a snapshot of the entire stack, and they return information representing the entire stack. There is no way to avoid the cost of examining all the frames if the caller is only interested in the top few frames on the stack. Both the `Throwable::getStackTrace` and

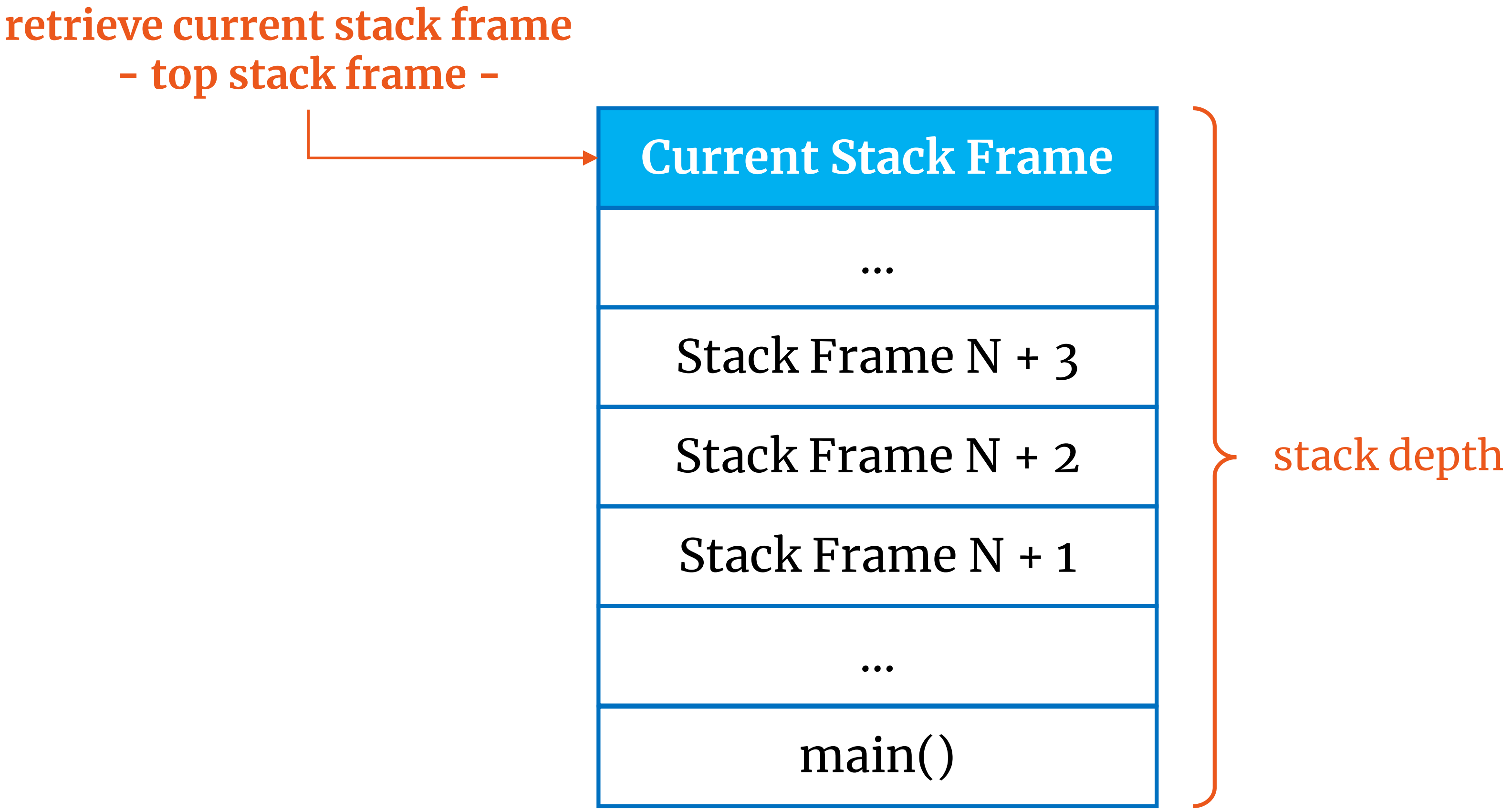
Stack-Walker - flexible mechanism to traverse and materialize the required stack-frame information allowing efficient lazy access to additional stack frames when required.

It avoids the cost of examining all frames if caller is interested in a few.

Stack-Walker features:

✓ lazy frames construction
✓ limit stack depth
✓ filter frames

```
public static StackWalker getInstance()

public static StackWalker getInstance(Set<Option> options)
      Option:
            // supports getCallerClass() and getDeclaringClass()
            RETAIN_CLASS_REFERENCE
            // shows all reflection frames and other hidden frames
            // that are implementation-specific.
            SHOW_REFLECT_FRAMES
            // shows all hidden frames (including reflection frames).
            SHOW_HIDDEN_FRAMES

public static StackWalker getInstance(Set<Option> options, int estimateDepth)
```


retrieve current stack frame
- top stack frame -
Current Stack Frame
...
Stack Frame N + 3
Stack Frame N + 2
Stack Frame N + 1
...
main()
stack depth

```java
@Benchmark
public StackWalker.StackFrame get_top_stack_frame() {
    return recTopStackFrame(stackDepth);
}
private StackWalker.StackFrame recTopStackFrame(int depth) {
    if (depth == 0) {
        return getTopStackFrame();
    }
    return recTopStackFrame(depth - 1);
}
private StackWalker.StackFrame getTopStackFrame() {
    return StackWalker.getInstance()
            .walk(stream -> stream.findFirst())
            .orElseThrow(NoSuchElementException::new);
}
```

```java
@Benchmark
public StackWalker.StackFrame get_top_stack_frame() {
    return recTopStackFrame(stackDepth);
}
private StackWalker.StackFrame recTopStackFrame(int depth) {
    if (depth == 0) {
        return getTopStackFrame();
    }
    return recTopStackFrame(depth - 1);
}
private StackWalker.StackFrame getTopStackFrame() {
    return StackWalker.getInstance()
            .walk(stream -> stream.findFirst())
            .orElseThrow(NoSuchElementException::new);
}
```

		Average Time (us/op)
Stack Depth	**StackWalker**	**getStackTrace()**
1	1.397 ± 0.074	31.029 ± 1.812
10	1.347 ± 0.066	47.933 ± 7.189
100	1.482 ± 0.078	166.431 ± 9.379
1000	3.202 ± 0.157	1,446.025 ± 71.891

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

		Average Time (us/op)
Stack Depth	**StackWalker**	**getStackTrace()**
1	1.397 ± 0.074	31.029 ± 1.812
10	1.347 ± 0.066	47.933 ± 7.189
100	1.482 ± 0.078	166.431 ± 9.379
1000	3.202 ± 0.157	1,446.025 ± 71.891

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

- **StackWalker** offers almost constant access to current/top frame for different StackDepth sizes.
- **getStackTrace()** cost depends on StackDepth size and is at least one order of magnitude slower.

Performance Scenario:

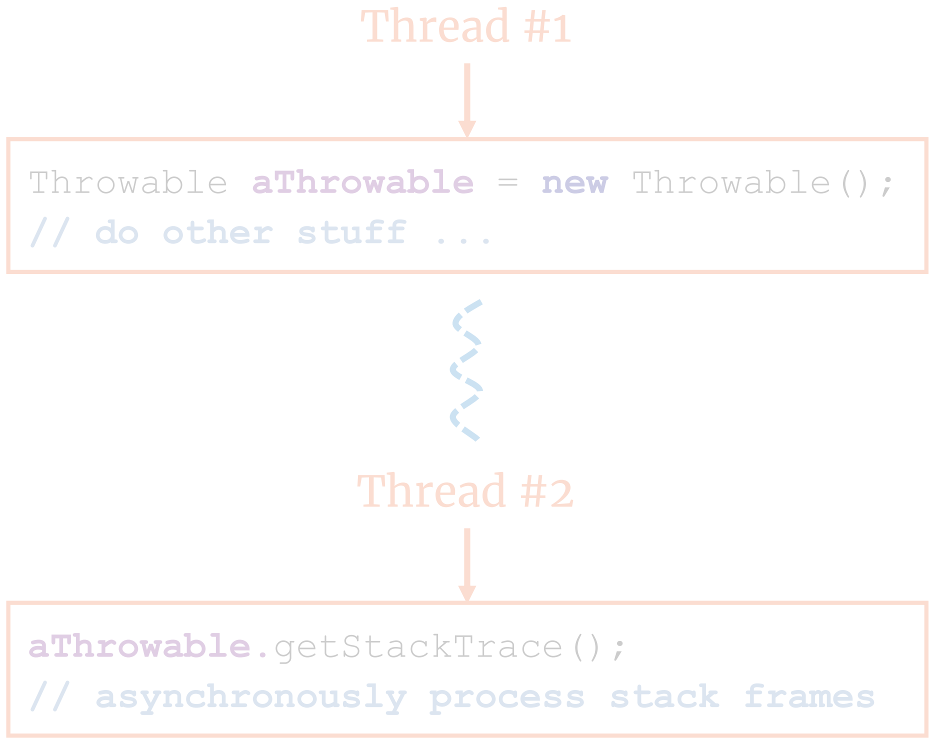
process asynchronously throwable frames (in another thread) vs. [synchronously] stack walking

Thread #1

```
Throwable aThrowable = new Throwable();
// do other stuff ...
```

Thread #2

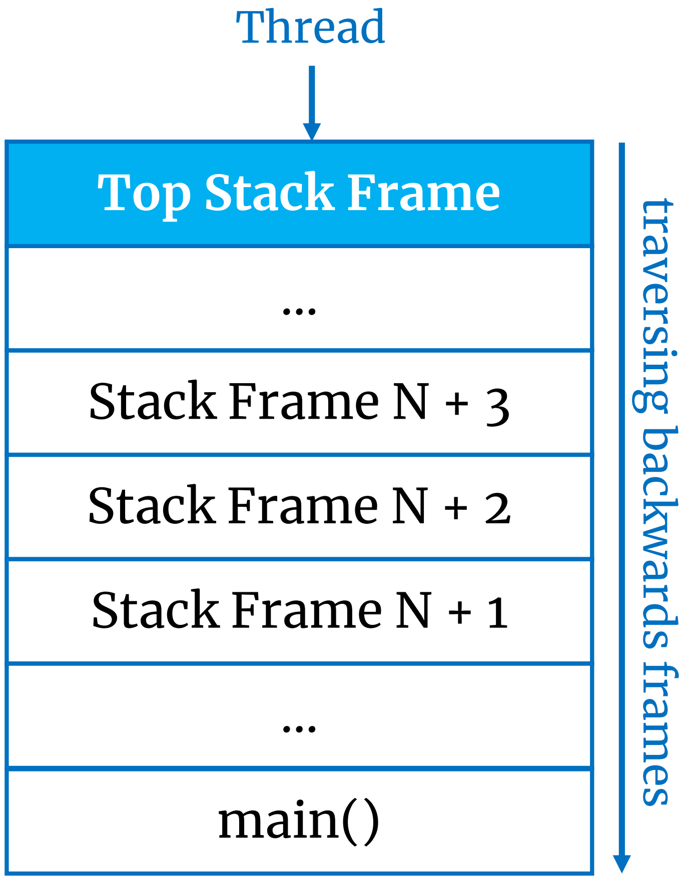
```
aThrowable.getStackTrace();
// asynchronously process stack frames
```



```
Throwable aThrowable = new Throwable();
// do other stuff ...
```

```
aThrowable.getStackTrace();
// asynchronously process stack frames
```

vs.



		Average Time (us/op)
Stack Depth	**StackWalker** [1]	**new Throwable()** [2]
1	1.411 ± 0.079	1.186 ± 0.064
10	4.647 ± 0.215	1.518 ± 0.075
100	26.909 ± 1.412	5.199 ± 0.230
1000	250.267 ± 13.402	40.944 ± 2.089

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

[1] – includes the cost of traversing synchronously (backwards) stack frames.
[2] – excludes the cost of generating stack trace elements (i.e. getStackTrace()).

		Average Time (us/op)
Stack Depth	**StackWalker** [1]	**new Throwable()** [2]
1	1.411 ± 0.079	1.186 ± 0.064
10	4.647 ± 0.215	1.518 ± 0.075
100	26.909 ± 1.412	5.199 ± 0.230
1000	250.267 ± 13.402	40.944 ± 2.089

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

[1] – includes the cost of traversing synchronously (backwards) stack frames.
[2] – excludes the cost of generating stack trace elements (i.e. getStackTrace()).

		Average Time (us/op)
Stack Depth	StackWalker [1]	new Throwable() [2]
1	1.411 ± 0.0[illegible]	1.186 ± 0.064
10	[illegible]	[illegible]
100	26.909 ± 1.412	5.199 ± 0.230
1000	250.267 ± 13.462	40.944 ± 2.089

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

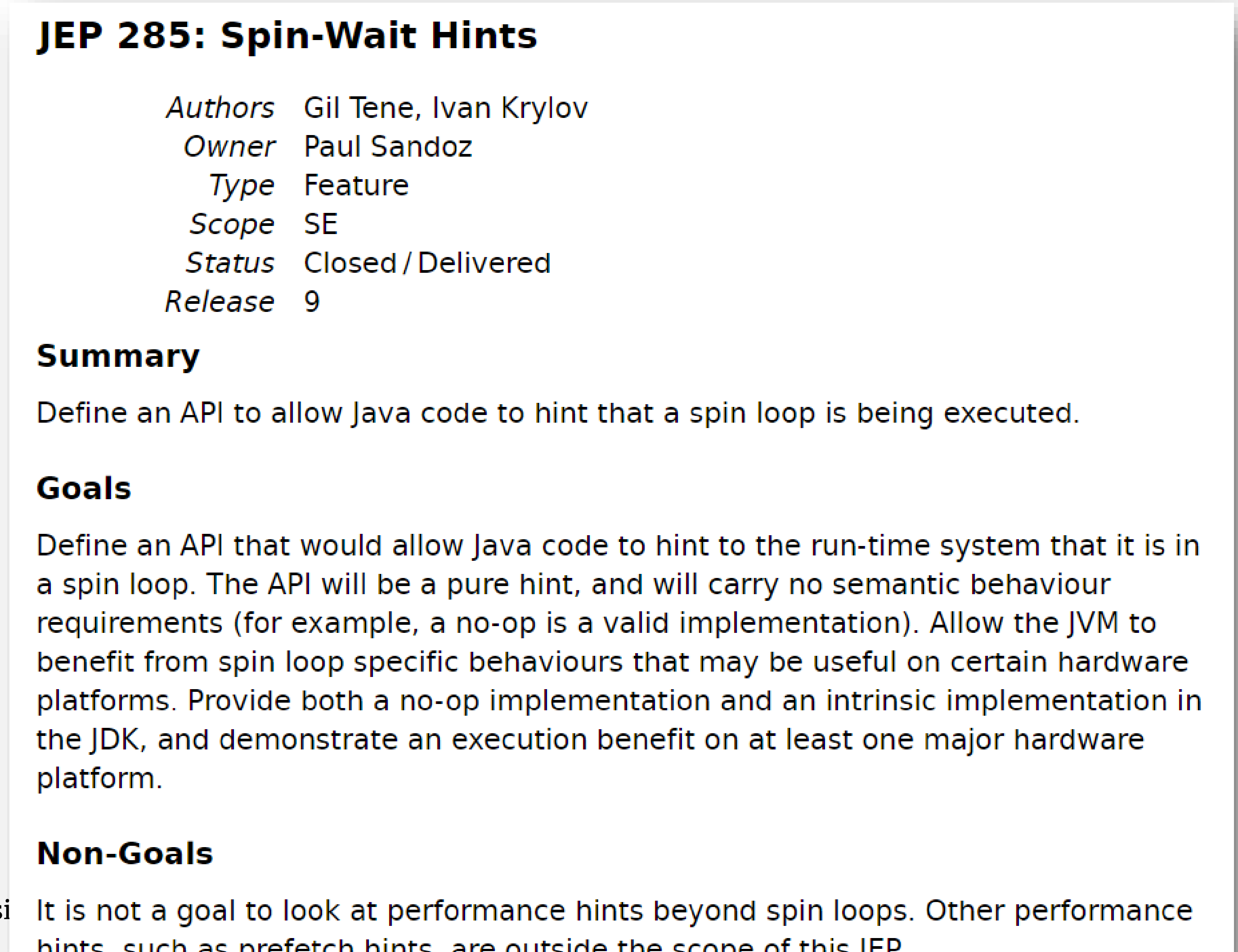
Processing asynchronously throwable frames (i.e. in another thread) is even faster than Stack Walker API.

Advice: leverage to this mechanism when affordable, for a better performance!

[1] – includes the cost of synchronously traversing (backwards) N stack frames
[2] – excludes the cost of generating stack trace elements (i.e. getStackTrace())

Spin-Wait Hints

05



JEP 285: Spin-Wait Hints

Authors	Gil Tene, Ivan Krylov
Owner	Paul Sandoz
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9

Summary

Define an API to allow Java code to hint that a spin loop is being executed.

Goals

Define an API that would allow Java code to hint to the run-time system that it is in a spin loop. The API will be a pure hint, and will carry no semantic behaviour requirements (for example, a no-op is a valid implementation). Allow the JVM to benefit from spin loop specific behaviours that may be useful on certain hardware platforms. Provide both a no-op implementation and an intrinsic implementation in the JDK, and demonstrate an execution benefit on at least one major hardware platform.

Non-Goals

It is not a goal to look at performance hints beyond spin loops. Other performance hints, such as prefetch hints, are outside the scope of this JEP onSpinWait() - tells the CPU there is busy-waiting loop that may burn few CPU-cycles waiting for something to happen.

CPU can assign more resources to other threads, without actually invoking the OS scheduler to dequeue the thread (which may be expensive).

```
while (!condition_not_satisfied) {
        Thread.onSpinWait();
}
```

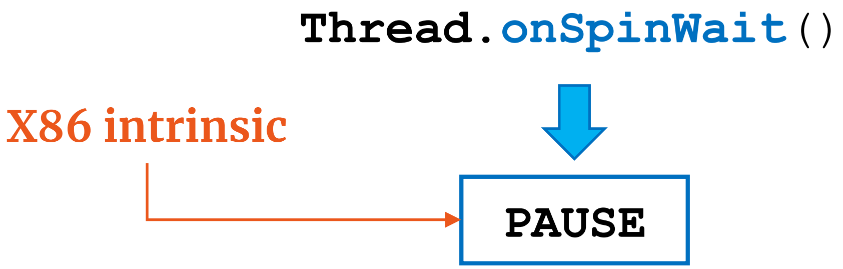
onSpinWait() – suitable for events which might happen very frequently and finish very quickly (with a higher rate).

Producer

```
for (long i = 0; i < items; i++) {
    while (!ready_to_produce) {
        Thread.onSpinWait();
    }
    produce_item();
}
// signal is_running = false
signal_finish();
```

Consumer

```
while (is_running) {
    while (!ready_to_consume) {
        Thread.onSpinWait();
    }
    consume_item();
}
```



PAUSE instruction delays the next instruction's execution for a finite period of time, hence parts of the pipeline are no longer being used → *reduces power consumption*!

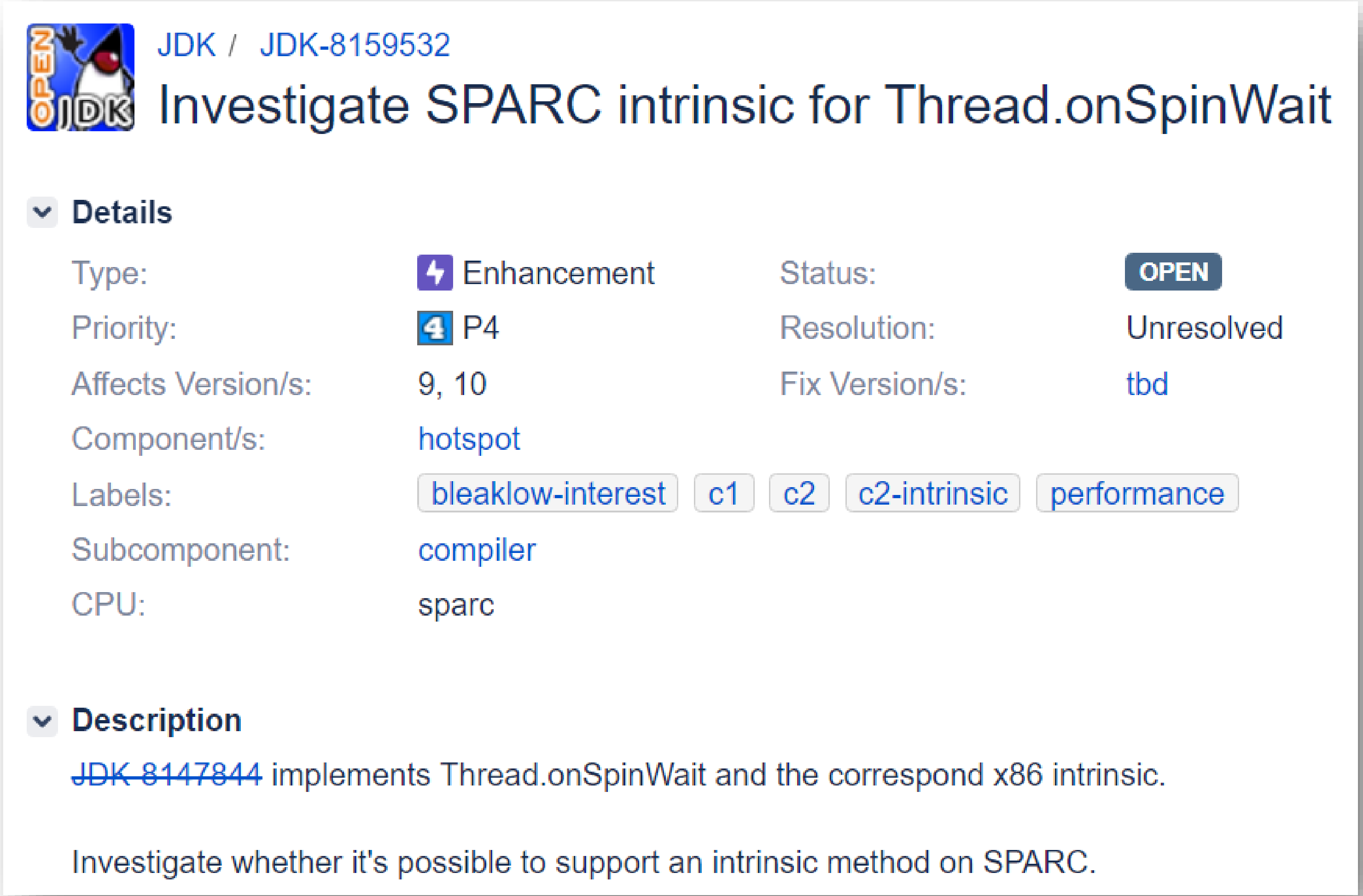
Source: [https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops]

	Average Time (us/op)	Context Switches [1]
onSpinWait()	39.419 ± 0.215	2,815
yield()	321.749 ± 5.175	2,826
parkNanos(1)	431.874 ± 16.510	2,821
sleep(1)	1,361,715.726 ± 31,258.272	18,504

[1] – difficult to report an accurate number of context switches!

	Average Time (us/op)	Context Switches [1]
onSpinWait()	39.419 ± 0.215	2,815
yield()	321.749 ± 5.175	2,826
parkNanos(1)	431.874 ± 16.510	2,831
sleep(1)	1,361,715.726 ± 31,258.272	18,504

[1] – difficult to report an accurate number of context switches!


JDK / JDK-8159532

Investigate SPARC intrinsic for Thread.onSpinWait

Details

Type:	Enhancement	Status:	OPEN
Priority:	P4	Resolution:	Unresolved
Affects Version/s:	9, 10	Fix Version/s:	tbd
Component/s:	hotspot		
Labels:	bleaklow-interest c1 c2 c2-intrinsic performance		
Subcomponent:	compiler		
CPU:	sparc		

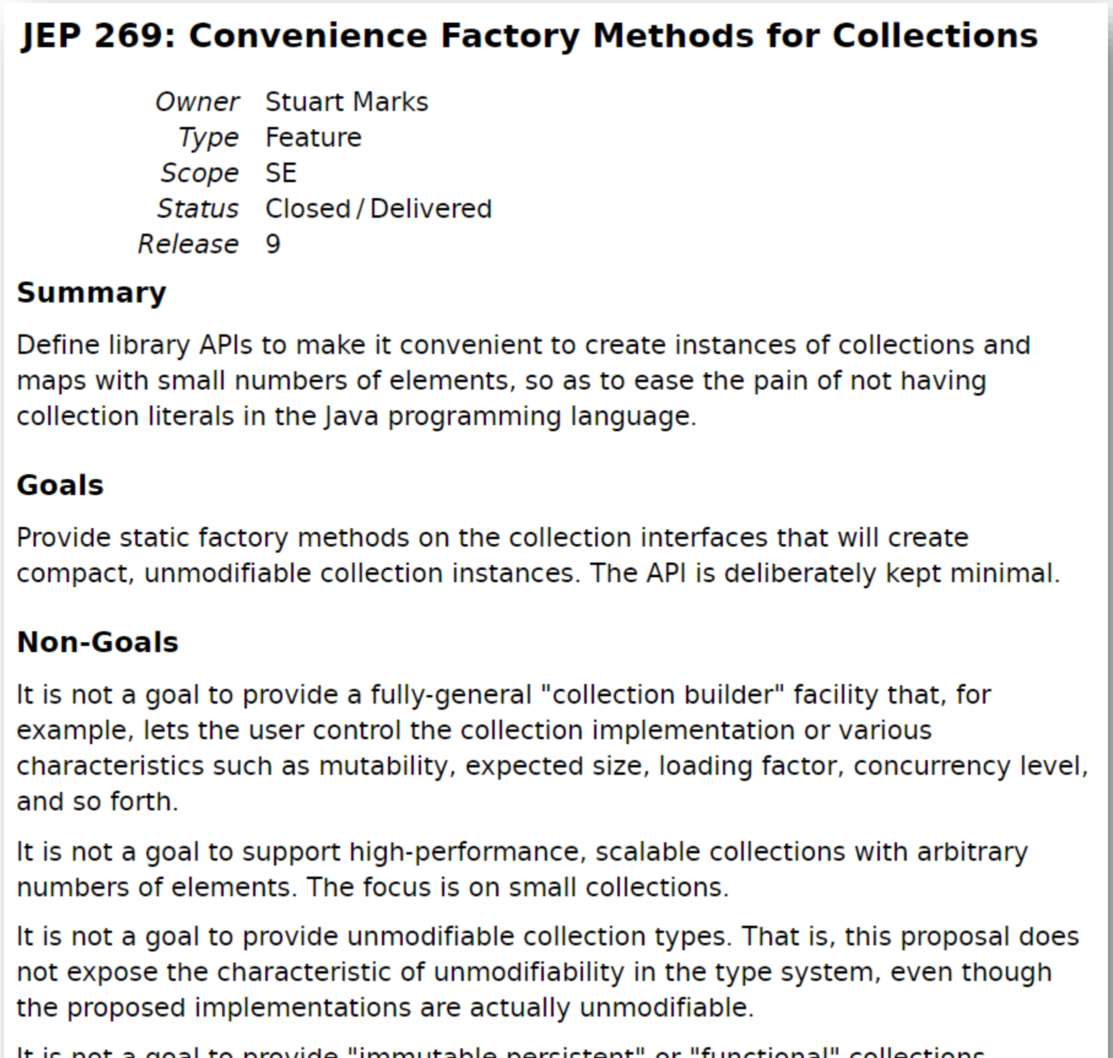
Description

~~JDK-8147844~~ implements Thread.onSpinWait and the correspond x86 intrinsic.

Investigate whether it's possible to support an intrinsic method on SPARC.

Collections Factory Methods

06



JEP 269: Convenience Factory Methods for Collections

Owner	Stuart Marks
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9

Summary

Define library APIs to make it convenient to create instances of collections and maps with small numbers of elements, so as to ease the pain of not having collection literals in the Java programming language.

Goals

Provide static factory methods on the collection interfaces that will create compact, unmodifiable collection instances. The API is deliberately kept minimal.

Non-Goals

It is not a goal to provide a fully-general "collection builder" facility that, for example, lets the user control the collection implementation or various characteristics such as mutability, expected size, loading factor, concurrency level, and so forth.

It is not a goal to support high-performance, scalable collections with arbitrary numbers of elements. The focus is on small collections.

It is not a goal to provide unmodifiable collection types. That is, this proposal does not expose the characteristic of unmodifiability in the type system, even though the proposed implementations are actually unmodifiable.

It is not a goal to provide "immutable persistent" or "functional" collections

Factory Methods - static factory methods on the collection interfaces that creates **compact**, **unmodifiable** collection instances.

```
List.of("a", "b", "c")

Set.of("a", "b", "c")

Map.of("a", "v1", "b", "v2")
Map.ofEntries(entry("a", "v1"), entry("b", "v2"))
```

List.of()

explicit methods for ≤ 10 args

```
<E> List<E> of()
<E> List<E> of(E e1)
<E> List<E> of(E e1, E e2)
<E> List<E> of(E e1, E e2, E e3)
<E> List<E> of(E e1, E e2, E e3, E e4)
<E> List<E> of(E e1, E e2, E e3, E e4, E e5)
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6)
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
<E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

<E> List<E> of(E... elements) // var_args method
```

var_args(int ... args) // 4 args method call

prior calling var_args

```
mov DWORD PTR [rdx+0x8],0xf800016d      // {metadata({array int})}
mov DWORD PTR [rdx+0xc],0x4

                                        // Collect stack params
mov r10d,DWORD PTR [rbp+0x10]           //*getfield param2
mov r8d,DWORD PTR [rbp+0x14]            //*getfield param3
mov r11d,DWORD PTR [rbp+0x18]           //*getfield param4
mov ecx,DWORD PTR [rbp+0xc]             //*getfield param1

                                        // Add params to array
mov DWORD PTR [rdx+0x10],ecx            //*iastore
mov DWORD PTR [rdx+0x14],r10d           //*iastore
mov DWORD PTR [rdx+0x18],r8d            //*iastore
mov DWORD PTR [rdx+0x1c],r11d           //*newarray

call 0x00007f68e963e8c0                 //*invokespecial var_args
                                        // {optimized virtual_call}
```

```
var_args(int ... args)                        // 4 args method call

prior calling var_args

  mov DWORD PTR [rdx+0x8],0xf800016d          // {metadata({array int})}
  mov DWORD PTR [rdx+0xc],0x4
                                              // Collect stack params
  mov r10d,DWORD PTR [rbp+0x10]               //*getfield param2
  mov r8d,DWORD PTR [rbp+0x14]                //*getfield param3
  mov r11d,DWORD PTR [rbp+0x18]               //*getfield param4
  mov ecx,DWORD PTR [rbp+0xc]                 //*getfield param1

                                              // Add params to array
  mov DWORD PTR [rdx+0x10],ecx                //*iastore
  mov DWORD PTR [rdx+0x14],r10d               //*iastore
  mov DWORD PTR [rdx+0x18],r8d                //*iastore
  mov DWORD PTR [rdx+0x1c],r11d               //*newarray

  call 0x00007f68e963e8c0                     //*invokespecial var_args
                                              // {optimized virtual_call}
```

```
var_args(int ... args)                              // 4 args method call

    mov DWORD PTR [rdx+0x8],0xf800016d              // {metadata({array int})}
    mov DWORD PTR [rdx+0xc],0x4
                                                    // COLLECT STACK PARAMETERS
    mov r10d,DWORD PTR [rbp+0x10]                   //*getfield param2
    mov r8d,DWORD PTR [rbp+0x14]                    //*getfield param3
    mov r11d,DWORD PTR [rbp+0x18]                   //*getfield param4
    mov ecx,DWORD PTR [rbp+0xc]                     //*getfield param1

                                                    // STORE PARAMETERS TO ARRAY
    mov DWORD PTR [rdx+0x10],ecx                    //*iastore
    mov DWORD PTR [rdx+0x14],r10d                   //*iastore
    mov DWORD PTR [rdx+0x18],r8d                    //*iastore
    mov DWORD PTR [rdx+0x1c],r11d                   //*newarray

    mov rsi,rbp
    call 0x00007f68e963e8c0                         //*invokespecial var_args
                                                    // {optimized virtual_call}
```

prior calling var_args

var_args method calls deal with extra stack parameter manipulation and array allocation (i.e. impacting the performance)

		Average Time (us/op)
No. of args	**explicit method params**	**var_args method**
2	2.304 ± 0.097	4.367 ± 0.364
4	2.575 ± 0.022	4.958 ± 0.067
6	3.015 ± 0.206	7.068 ± 0.812
8	3.342 ± 0.062	7.950 ± 1.151
10	3.748 ± 0.110	8.676 ± 0.606

Runtime: *[OpenJDK 11.0.2 Linux 64-bit]*

		Average Time (us/op)
No. of args	**explicit method params**	**var_args method**
2	2.304 ± 0.097	4.367 ± 0.364
4	2.575 ± 0.022	4.958 ± 0.067
6	3.015 ± 0.206	7.068 ± 0.812
8	3.342 ± 0.062	7.950 ± 1.151
10	3.748 ± 0.110	8.676 ± 0.606

***Runtime:** [OpenJDK 11.0.2 Linux 64-bit]*

```
static <E> List<E> of() {
    return ImmutableCollections.emptyList();
}

static <E> List<E> of(E e1) {
    return new ImmutableCollections.List12<>(e1);
}

static <E> List<E> of(E e1, E e2) {
    return new ImmutableCollections.List12<>(e1, e2);
}

static <E> List<E> of(E e1, E e2, E e3) {
    return new ImmutableCollections.ListN<>(e1, e2, e3);
}
```

custom wrappers for 1 and 2 args

custom wrapper for N > 2 args

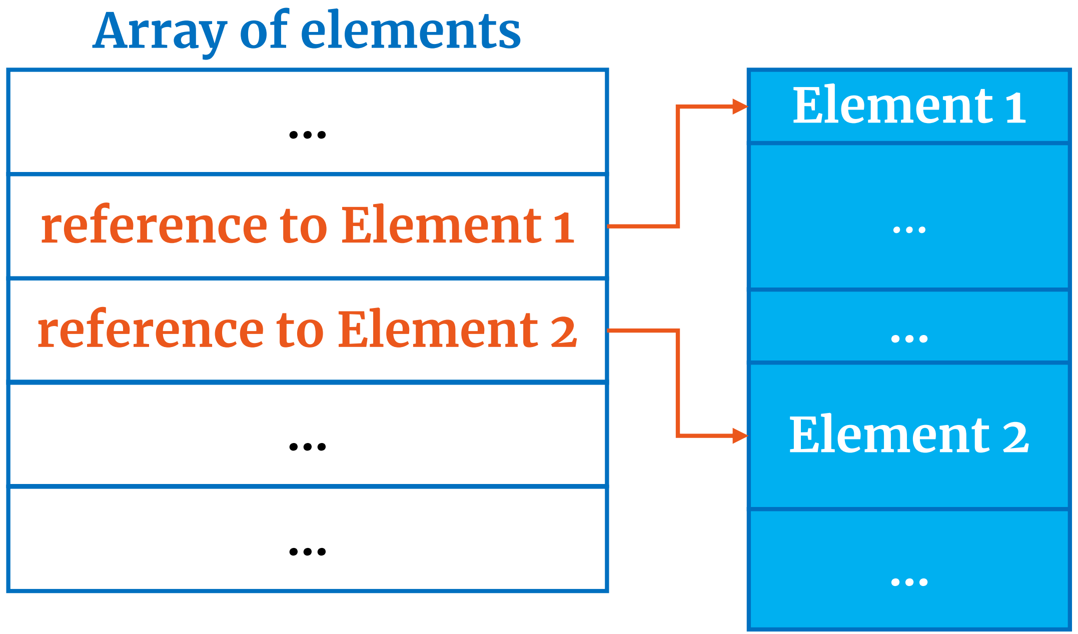
```
static final class List12<E> extends AbstractImmutableList<E> {
    @Stable private final E e0;
    @Stable private final E e1;
    // ...
}
```

packed elements implementation

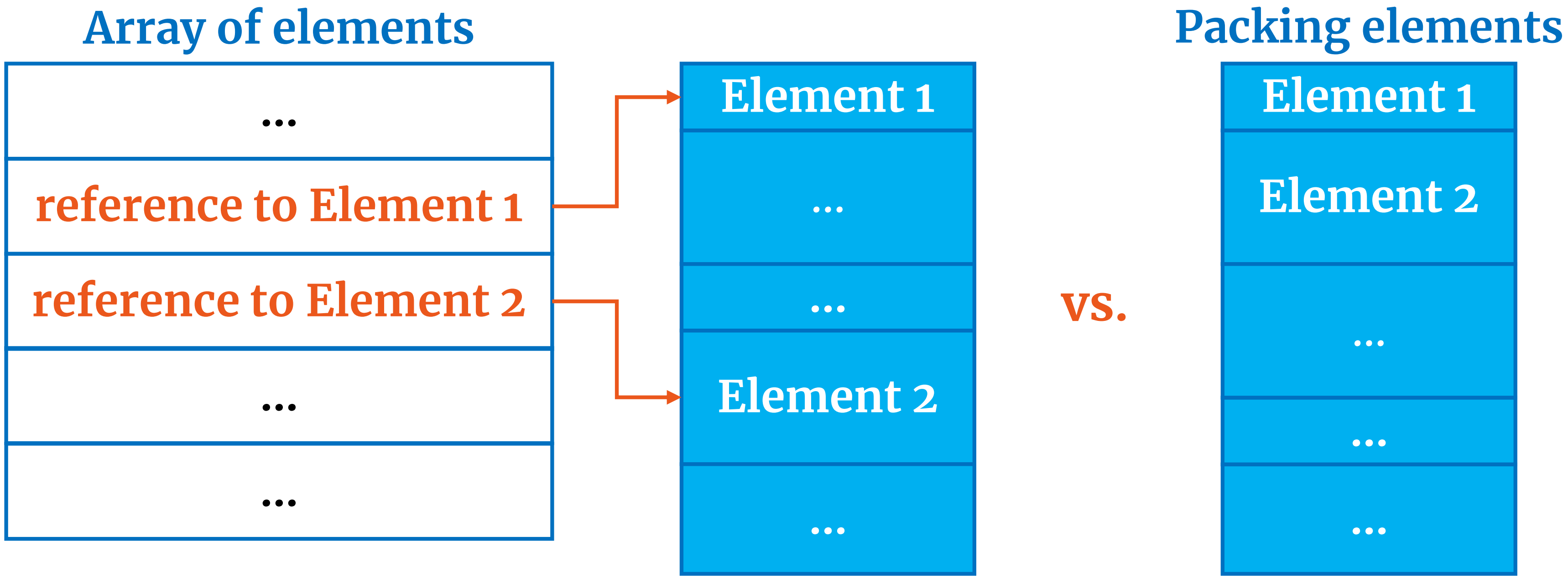
```
static final class ListN<E> extends AbstractImmutableList<E> {
    @Stable private final E[] elements;
    // ...
}
```

array based implementation

Data Locality



Data Locality



Packing elements reduces the number of indirections and improves data locality (i.e. minimize CPU D-Cache misses)

In a nutshell:

reduced memory footprint

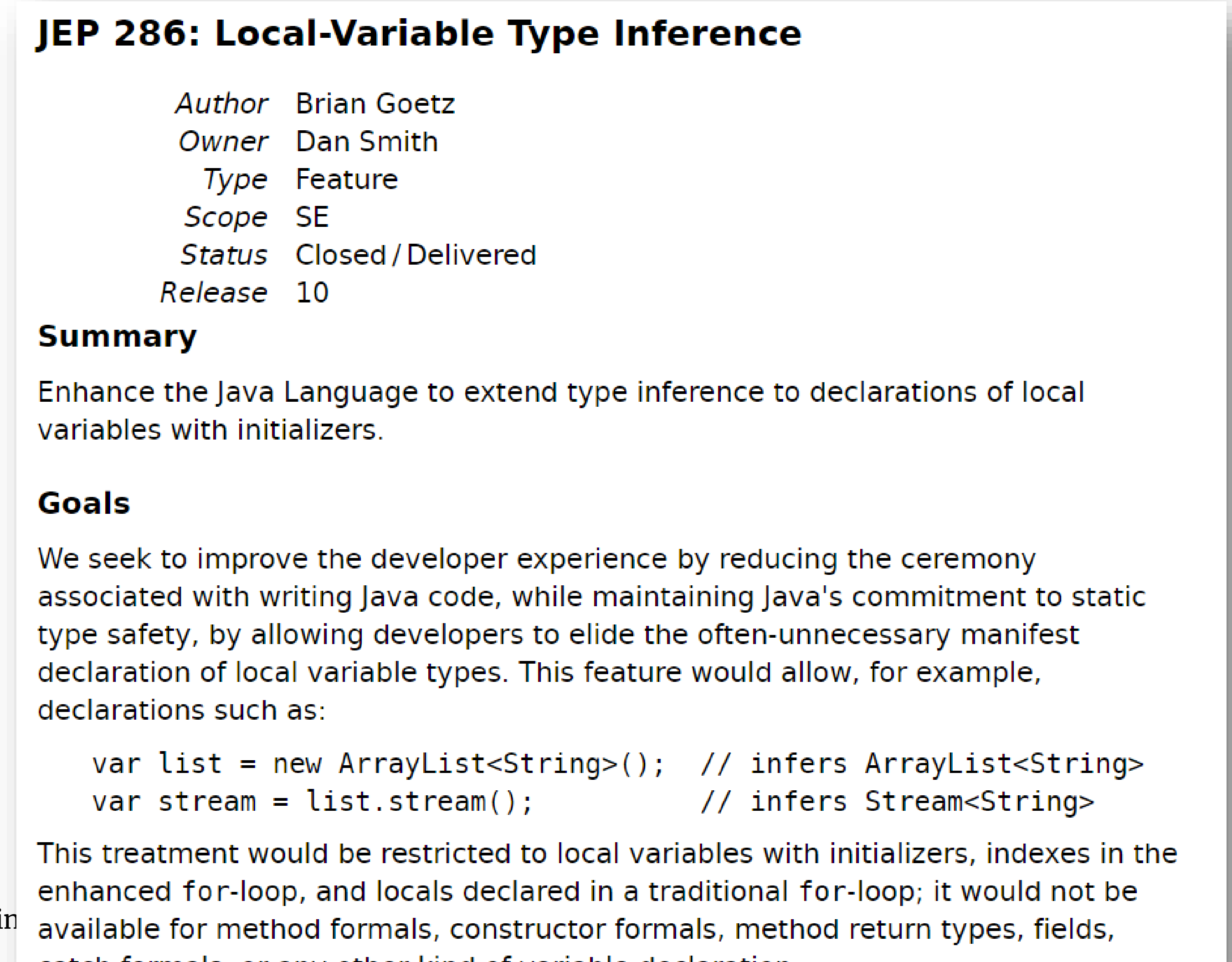
improved data locality

Set.of() and **Map.of()** are implemented based on similar design principles:

{ custom wrappers, explicit method parameters }

Local-Variable Type Inference

07



JEP 286: Local-Variable Type Inference

Author	Brian Goetz
Owner	Dan Smith
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	10

Summary

Enhance the Java Language to extend type inference to declarations of local variables with initializers.

Goals

We seek to improve the developer experience by reducing the ceremony associated with writing Java code, while maintaining Java's commitment to static type safety, by allowing developers to elide the often-unnecessary manifest declaration of local variable types. This feature would allow, for example, declarations such as:

```
var list = new ArrayList<String>();  // infers ArrayList<String>
var stream = list.stream();          // infers Stream<String>
```

This treatment would be restricted to local variables with initializers, indexes in the enhanced `for`-loop, and locals declared in a traditional `for`-loop; it would not be available for method formals, constructor formals, method return types, fields,

```java
var aBool = true;
var aChar = '\ufffd';
var aLong = 0L;
var aString = "conference";
var aFloat = 1.0f;
var aDouble = 2.0;
var listOf = List.of("a", "b");
var aBool = true;
var aChar = '\ufffd';
var aLong = 0L;
var aString = "conference";
var aFloat = 1.0f;
var aDouble = 2.0;
var listOf = List.of("a", "b");
```

GENERATED CODE

```java
boolean aBool = true;
char aChar = 65533;
long aLong = 0L;
String aString = "conference";
float aFloat = 1.0F;
double aDouble = 2.0D;
List<String> listOf = List.of("a", "b");
```

var is only syntactic sugar,
it does not exist at bytecode level.

```java
var aByte = 0;
var aShort = 0x7fff;  // intended as short
var anotherLong = 17; // intended as long
```

GENERATED CODE

```java
int aByte = 0;
int aShort = 32767;   // intended as short but widened to int!
int anotherLong = 17; // intended as long but narrowed to int!
```

When the initializer is a numeric value (especially for integer literals) without an explicit compiler hint, that numeric value may be silently widened or narrowed (e.g. inferred implicitly as an **int**).

```
var aList = new ArrayList<>(); // diamond operator

var anotherList = arrayToList(anArray); // generic method call
public static <T> List<T> arrayToList(String[] a) {...}
```

GENERATED CODE

```
ArrayList<Object> aList = new ArrayList();

List<Object> anotherList = arrayToList(anArray);

// the inferred type cannot be deduced, it falls back to Object!
```

In case of **diamond operator** or **generic methods** where there is no explicit target type neither the actual constructor/method arguments to provide sufficient type information, the value will be inferred as <**Object**>.

Non-denotable types

{ anonymous class type, intersection type, capture variable type, NULL type }

Non-denotable types are types internally used by the compiler, which cannot be explicitly written out in the program source code.

```java
var obj = new Object() { // obj has a non-denotable type
    public void foo() {
        System.out.println("Java Conference!");
    }
};
```

```java
var obj = new Object() { // obj has a non-denotable type
    public void foo() {
        System.out.println("Java Conference!");
    }
};
```

BYTECODE

```
new NonDenotableTypeMain$1 // new type generated by javac
dup
invokespecial NonDenotableTypeMain$1.<init> ()V
astore 1
```

```
obj.foo(); // call not possible without type inference!
```

BYTECODE

```
aload 1
invokevirtual NonDenotableTypeMain$1.foo ()V
```

Type inference becomes very handy in case of **non-denotable types**, making the language more flexible.

Nest-Based Access Control

08

```java
public class Outer {

    private void prv_foo() {
        System.out.println("private foo method");
    }

    public class Inner {
        public void foo() {
            prv_foo(); // private method call
        }
    }
}
```

JAVA 10 GENERATED CODE

```
// Outer.class
public class Outer {
    public Outer();
    static void access$000(Outer) { // bridge method - NEW!
        aload_0
        invokespecial #1 // calls prv_foo:()V
    }
    private void prv_foo();
}
// Outer$Inner.class
public class Inner {
    final Outer this$0;
    public Inner(Outer);
    public void foo() {
        aload_0
        getfield      #1  // field this$0:LOuter;
        invokestatic  #3  // calls Outer.access$000:(LOuter;)V
    }
}
```

JAVA 10 GENERATED CODE

```
// Outer.class
public class Outer {
    public Outer();
    static void access$000(Outer) { // bridge method - NEW!
        aload_0
        invokespecial #1 // calls prv_foo:()V
    }
    private void prv_foo();
}
// Outer$Inner.class
public class Inner {
    final Outer this$0;
    public Inner(Outer);
    public void foo() {
        aload_0
        getfield      #1 1 // field this$0:LOuter;
        invokestatic  #3   // calls Outer.access$000:(LOuter;)V
    }
}
```

JAVA 10 GENERATED CODE

```
// Outer.class
public class Outer {
    public Outer();
    static void access$000(Outer) { // bridge method - NEW!
        aload_0
        invokespecial #1 // calls prv_foo:()V  2
    }
    private void prv_foo();
}
// Outer$Inner.class
public class Inner {
    final Outer this$0;
    public Inner(Outer);
    public void foo() {
        aload_0
        getfield      #1  1 // field this$0:LOuter;
        invokestatic  #3  // calls Outer.access$000:(LOuter;)V
    }
}
```

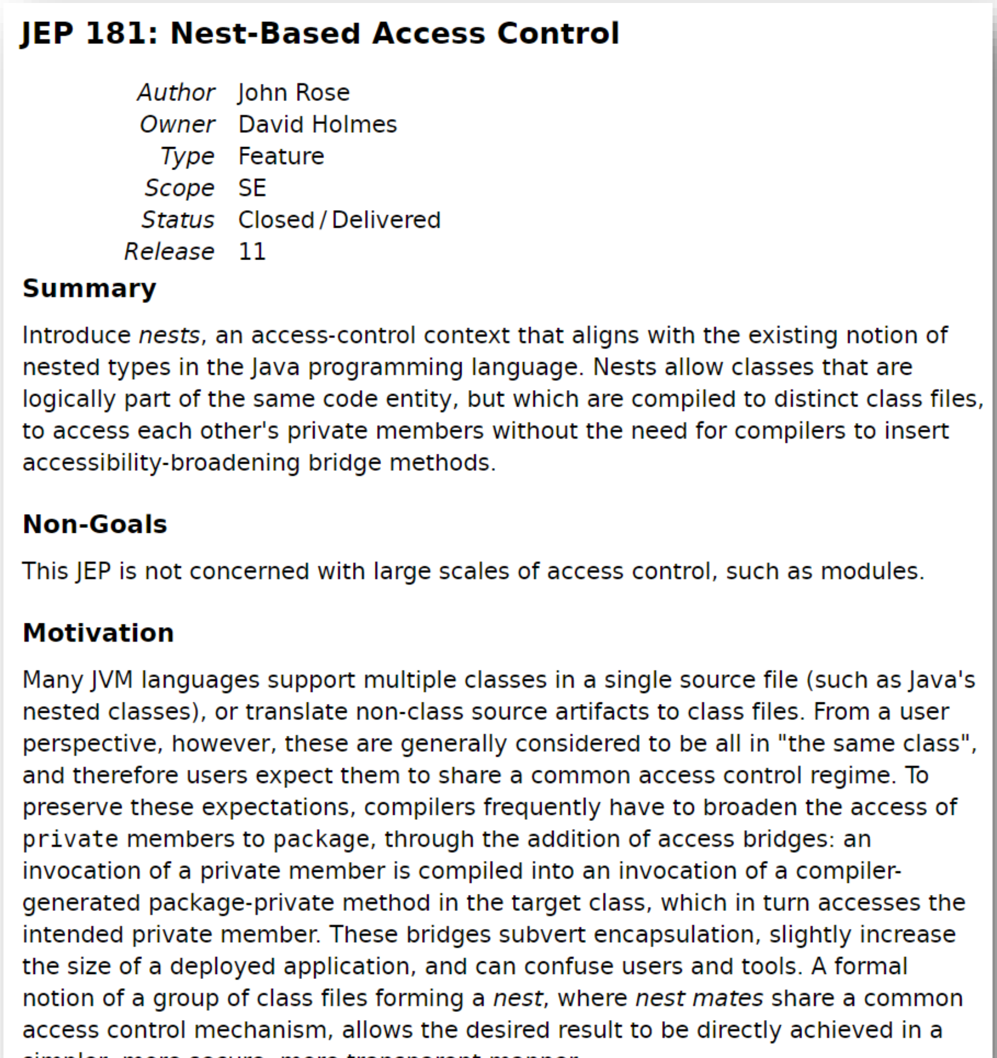
Potential drawbacks in case of bridges methods:

subvert encapsulation

slightly increase generated code size

can confuse users and tools

additional layer of indirection


JEP 181: Nest-Based Access Control

Author	John Rose
Owner	David Holmes
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	11

Summary

Introduce *nests*, an access-control context that aligns with the existing notion of nested types in the Java programming language. Nests allow classes that are logically part of the same code entity, but which are compiled to distinct class files, to access each other's private members without the need for compilers to insert accessibility-broadening bridge methods.

Non-Goals

This JEP is not concerned with large scales of access control, such as modules.

Motivation

Many JVM languages support multiple classes in a single source file (such as Java's nested classes), or translate non-class source artifacts to class files. From a user perspective, however, these are generally considered to be all in "the same class", and therefore users expect them to share a common access control regime. To preserve these expectations, compilers frequently have to broaden the access of `private` members to package, through the addition of access bridges: an invocation of a private member is compiled into an invocation of a compiler-generated package-private method in the target class, which in turn accesses the intended private member. These bridges subvert encapsulation, slightly increase the size of a deployed application, and can confuse users and tools. A formal notion of a group of class files forming a *nest*, where *nest mates* share a common access control mechanism, allows the desired result to be directly achieved in a

NestMates = {Outer.class, Inner.class}

```
class Outer {

    class Inner {
        //...
        NestHost:Outer // class attribute
    }

    //...
    NestMembers:Inner // class attribute
}
```

Nests allow to access each other's private members without the need for compilers to insert accessibility broadening bridge methods.

JAVA 11 GENERATED CODE

```
// Outer.class
public class Outer {
    public Outer();
    private void prv_foo();
    NestMembers: Inner
}
// Outer$Inner.class
public class Inner {
    final Outer this$0;
    public Inner(Outer);
    public void foo() {
        aload_0
        getfield      #1 // field this$0:LOuter;
        invokevirtual #3 // calls Outer.prv_foo:()V
        return
    }
    NestHost: Outer
}
```

JAVA 11 GENERATED CODE

```
// Outer.class
public class Outer {
    public Outer();
    private void prv_foo();
    NestMembers: Inner
}
// Outer$Inner.class
public class Inner {
    final Outer this$0;
    public Inner(Outer);
    public void foo() {
        aload_0
        getfield      #1 // field this$0:LOuter;
        invokevirtual #3 // calls Outer.prv_foo:()V
        return
    }
    NestHost: Outer
}
```

direct private member access,
no bridge method needed anymore!

- Thanks
- Gracias
- Danke
- Merci

Resources

Slides

https://IonutBalosin.com/talks

Further Readings

https://openjdk.java.net/projects/amber/LVTIstyle.html

https://www.youtube.com/watch?v=wIyeOaitmWM - Compact Strings

IonutBalosin.com/training

@IonutBalosin

Bonus Slides

UTF16 String Size Constraints

```
new String(new char[<length>]); // new String(<utf16String>.toCharArray());
new String(new char[<length>]); // new String(<utf16String>.toCharArray());
```

⬇

```
// @see java.lang.StringUTF16.java

public static byte[] newBytesFor(int length) {
    // ...
    if (length > MAX_LENGTH) { // MAX_LENGTH = Integer.MAX_VALUE >> 1
        throw new OutOfMemoryError("UTF16 String size is " + length +
                                   ", should be less than " + MAX_LENGTH);
    }
    return new byte[length << 1];
}
```

```
new String(new char[<length>]); // new String(<utf16String>.toCharArray());
```

UTF16 Strings cannot have the size bigger than **Integer.MAX_VALUE >> 1** due to underlying **byte[]** array which is actually doubled.

This constraint is not applicable for (1) **LATIN1 Strings** or (2) **-XX:-CompactStrings**.

```
        throw new OutOfMemoryError("UTF16 String size is " + length +
                                   ", should be less than " + MAX_LENGTH);
```

List.of()

JDK 8

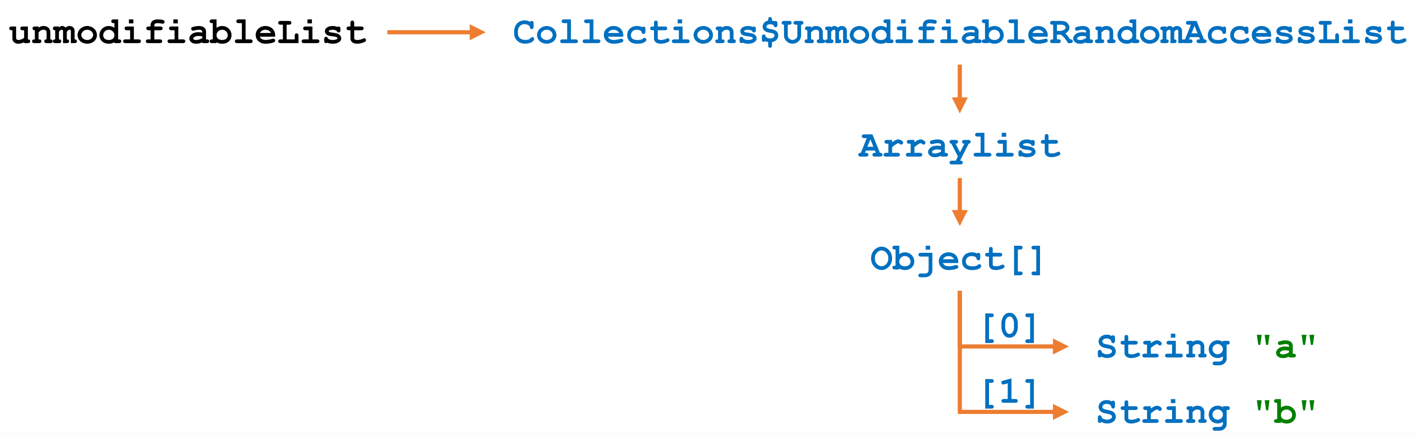
```
List<String> arrayList = new ArrayList<>();
arrayList.add("a");
arrayList.add("b");
List<String> unmodifiableList = Collections.unmodifiableList(arrayList);
```

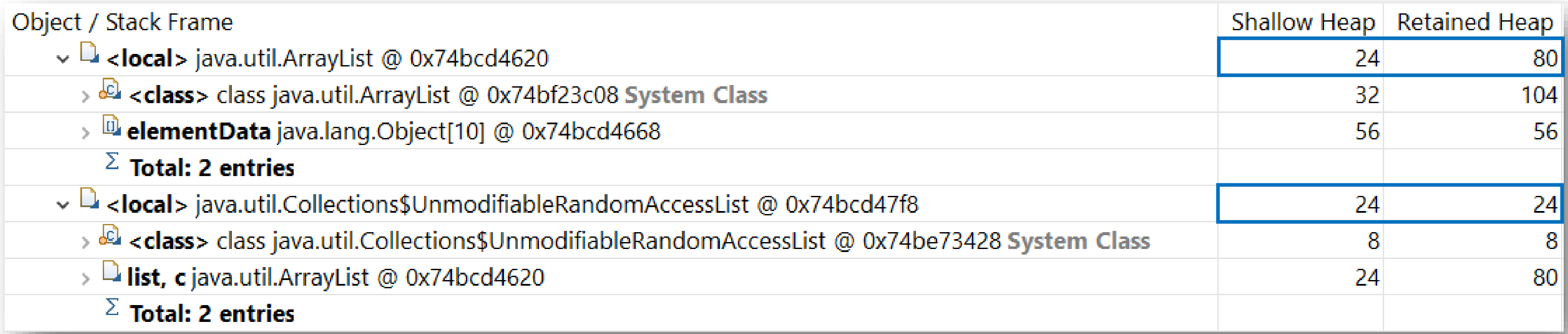
vs.

JDK 11

```
List<String> listOf = List.of("a", "b");
```

JDK 8

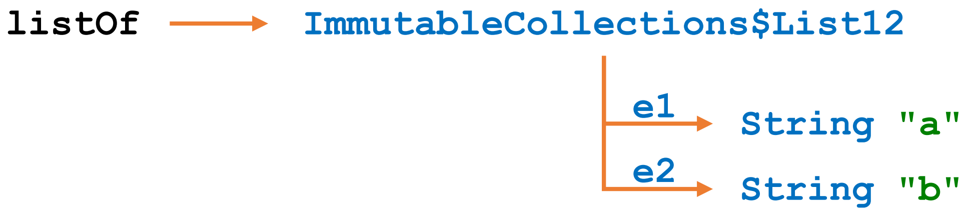


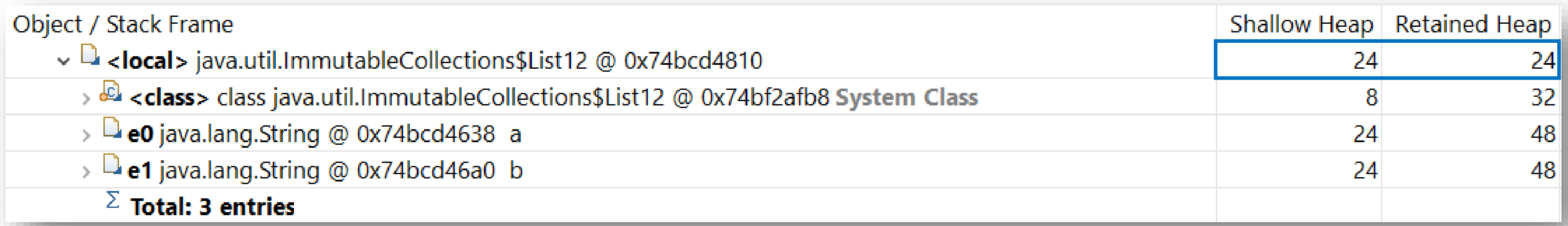


Object / Stack Frame	Shallow Heap	Retained Heap
<local> java.util.ArrayList @ 0x74bcd4620	24	80
<class> class java.util.ArrayList @ 0x74bf23c08 **System Class**	32	104
elementData java.lang.Object[10] @ 0x74bcd4668	56	56
Σ **Total: 2 entries**		
<local> java.util.Collections$UnmodifiableRandomAccessList @ 0x74bcd47f8	24	24
<class> class java.util.Collections$UnmodifiableRandomAccessList @ 0x74be73428 **System Class**	8	8
list, c java.util.ArrayList @ 0x74bcd4620	24	80
Σ **Total: 2 entries**		

Total Retained Heap: 80 + 24 bytes

JDK 11





Object / Stack Frame	Shallow Heap	Retained Heap
<local> java.util.ImmutableCollections$List12 @ 0x74bcd4810	24	24
<class> class java.util.ImmutableCollections$List12 @ 0x74bf2afb8 **System Class**	8	32
e0 java.lang.String @ 0x74bcd4638 a	24	48
e1 java.lang.String @ 0x74bcd46a0 b	24	48
Σ **Total: 3 entries**		

Total Retained Heap: 24 bytes

		Average Time (us/op)
No. of elements	**of()**	**unmodifiableList()**
empty	2.709 ± 0.162	8.349 ± 0.308
1	5.527 ± 0.283	19.051 ± 0.873
2	5.957 ± 0.258	23.931 ± 1.884
3	12.698 ± 0.549	27.078 ± 0.956
10	28.876 ± 1.675	46.579 ± 2.382
11	33.551 ± 1.849	67.406 ± 5.324

Runtime: [OpenJDK 11.0.2 Linux 64-bit]

Average Time
(us/op)

No. of elements	of()	unmodifiableList()
empty	2.709 ± 0.162	8.349 ± 0.308
1	5.527 ± 0.283	19.051 ± 0.873
2	5.957 ± 0.258	23.931 ± 1.884
3	12.698 ± 0.549	27.078 ± 0.956
10	28.876 ± 1.675	46.579 ± 2.382
11	33.551 ± 1.849	67.406 ± 5.324

***Runtime:** [OpenJDK 11.0.2 Linux 64-bit]*