What's New in HotSpot JVM 8

Vladimir Ivanov
HotSpot JVM Compiler team
Oracle Corp.

What's New in HotSpot JVM 8

Categorization

- Java 8 features support
  - Project Lambda, Nashorn, Type annotations
- Oracle implementation-specific improvements
  - won't be backported into 7
    - PermGen removal
  - will be/already backported into 7
    - numerous functional & performance enhancements

Project Lambda

Lambda expressions in Java

```
x -> x+1

(s,i) -> s.substring(0,i)

() -> System.out.print(“x”)

Predicate<String> pred = s -> s.length() < 100;
```

Project Lambda

Lambda expressions in Java

```
Function<Integer,Integer> f = x -> x+1
```

compiles to

```
invokedynamic [ j.l.i.LambdaMetafactory.metafactory,
                MethodType(Function.apply),
                MethodHandle(lambda$0) ] ()
```

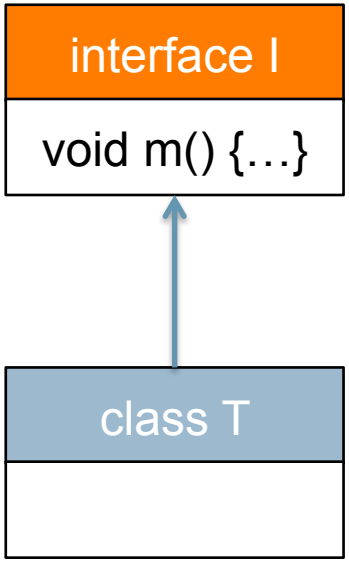
Project Lambda

Default methods

```
interface I {
    default void m() { /* do smth */ }
}

class T implements I {}

invokevirtual T.m()V => ?
```

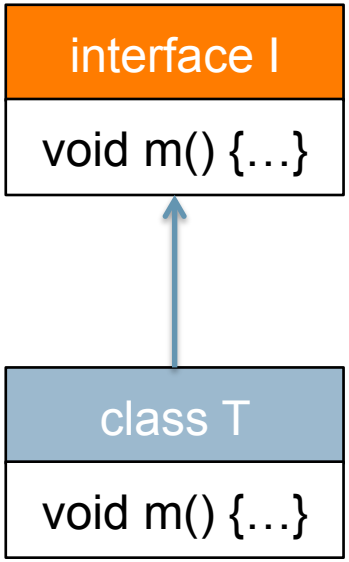


Project Lambda

Default methods

```
interface I {
    default void m() { /* do smth */ }
}
class T implements I {
    void m() { /* do smth */ }
}

invokevirtual T.m()V => ?
```

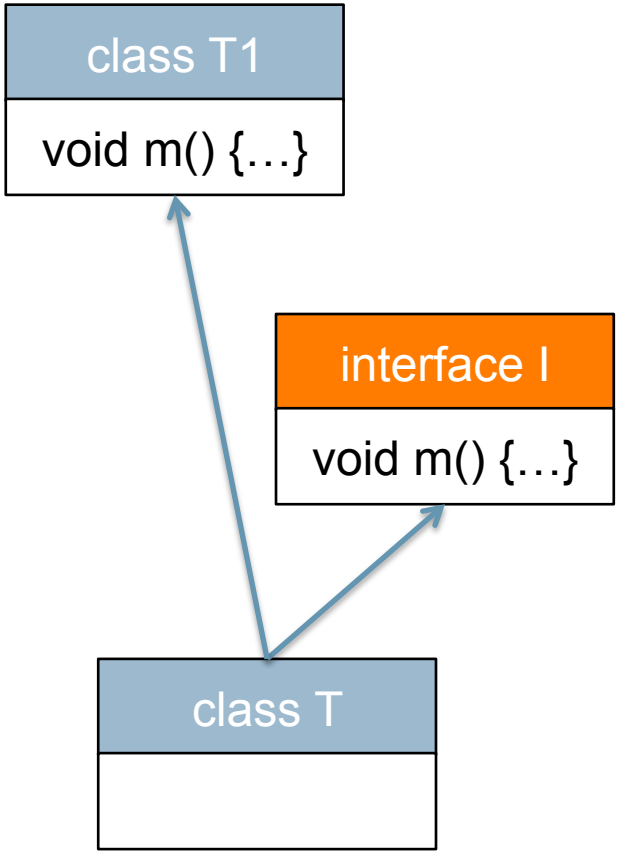


Project Lambda

Superclass overrides superinterface

```
interface I {
    default void m() { /* do smth */ }
}
class T1 implements I {
    void m() { /* do smth */ }
}
class T extends T1 implements I {}

invokevirtual T.m()V => ?
```

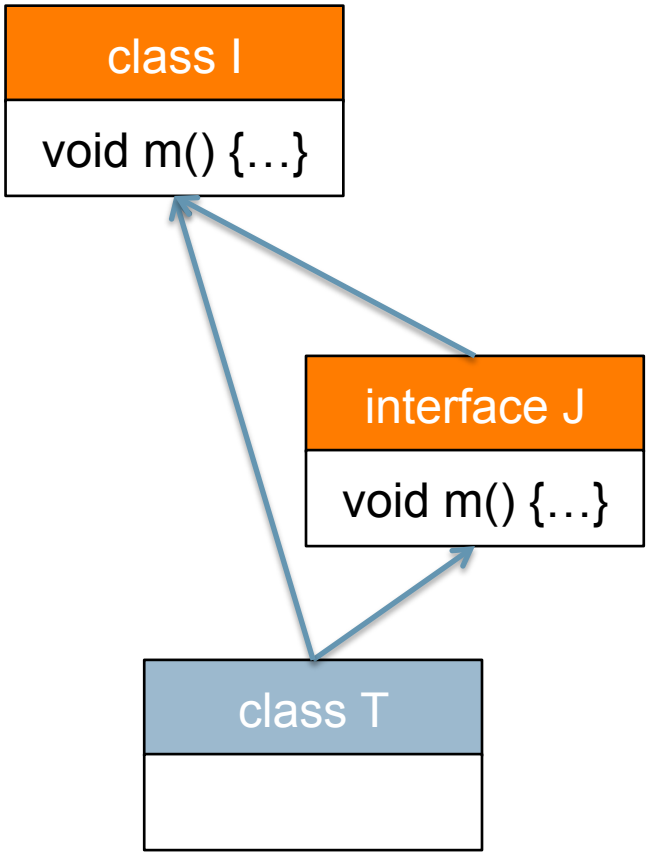


Project Lambda

Prefer most specific interface

```
interface I {
    default void m() { /* do smth */ }
}
interface J extends I {
    default void m() { /* do smth */ }
}
class T implements I,J {}

invokevirtual T.m()V => ?
```

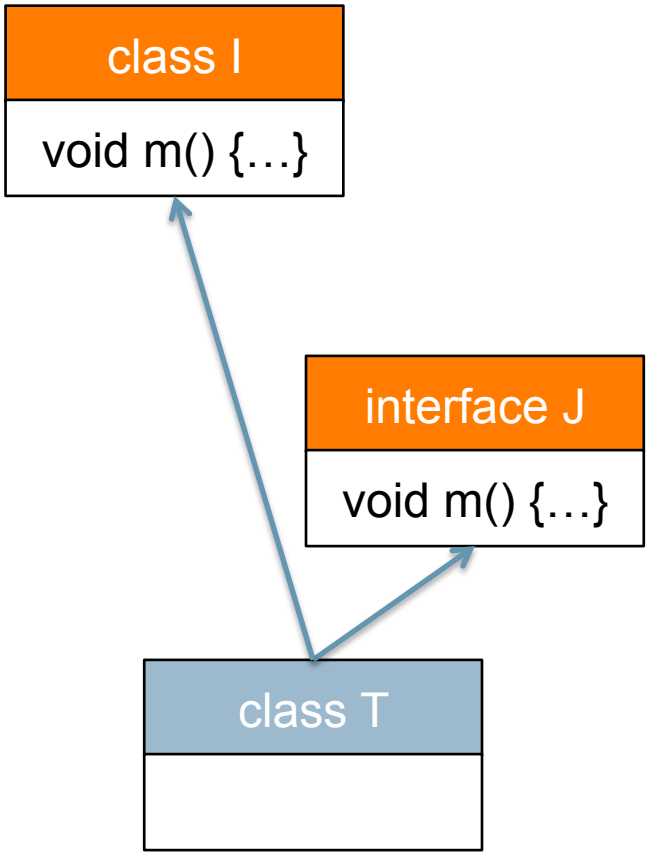


Project Lambda

Conflicting defaults

```
interface I {
    default void m() { /* do smth */ }
}
interface J {
    default void m() { /* do smth */ }
}
class T implements I,J {}

invokevirtual T.m()V => ?
```



Project Lambda

Static interface methods

```
interface I {
    static void m() { /* do smth */ }
}

invokestatic I.m();
```

Project Lambda

Private interface methods

```
interface I {
    private ... void m() { /* do smth */ }
}
```

Not allowed in Java.

Only on bytecode level.

Project Lambda

Bridge methods: Covariant overrides

```
class A                 { A get() {} }
class B extends A { B get() {} }
```

Javac adds the following:

```
synthetic bridge A get() { return ((B)this).get(); }
```

Project Lambda

Bridge methods: roads not taken

- Attempted to:
  - use invokedynamic
  - generate bridges by VM
- Finally:
  - continue to generate bridge methods by javac

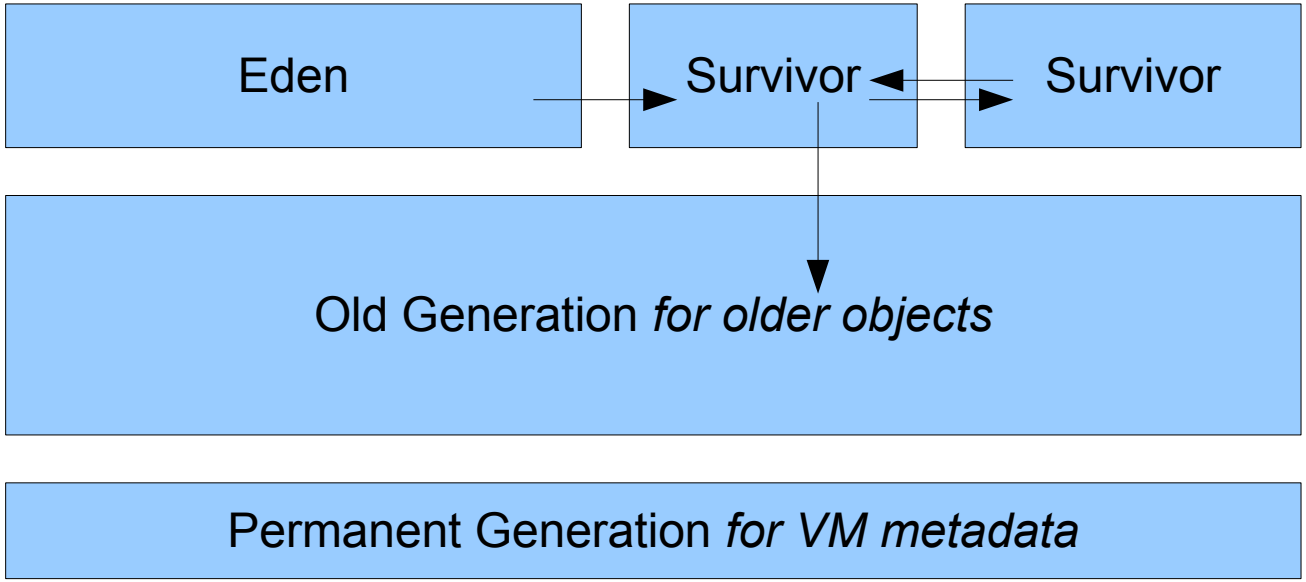

PermGen

java.lang.OutOfMemoryError: PermGen space

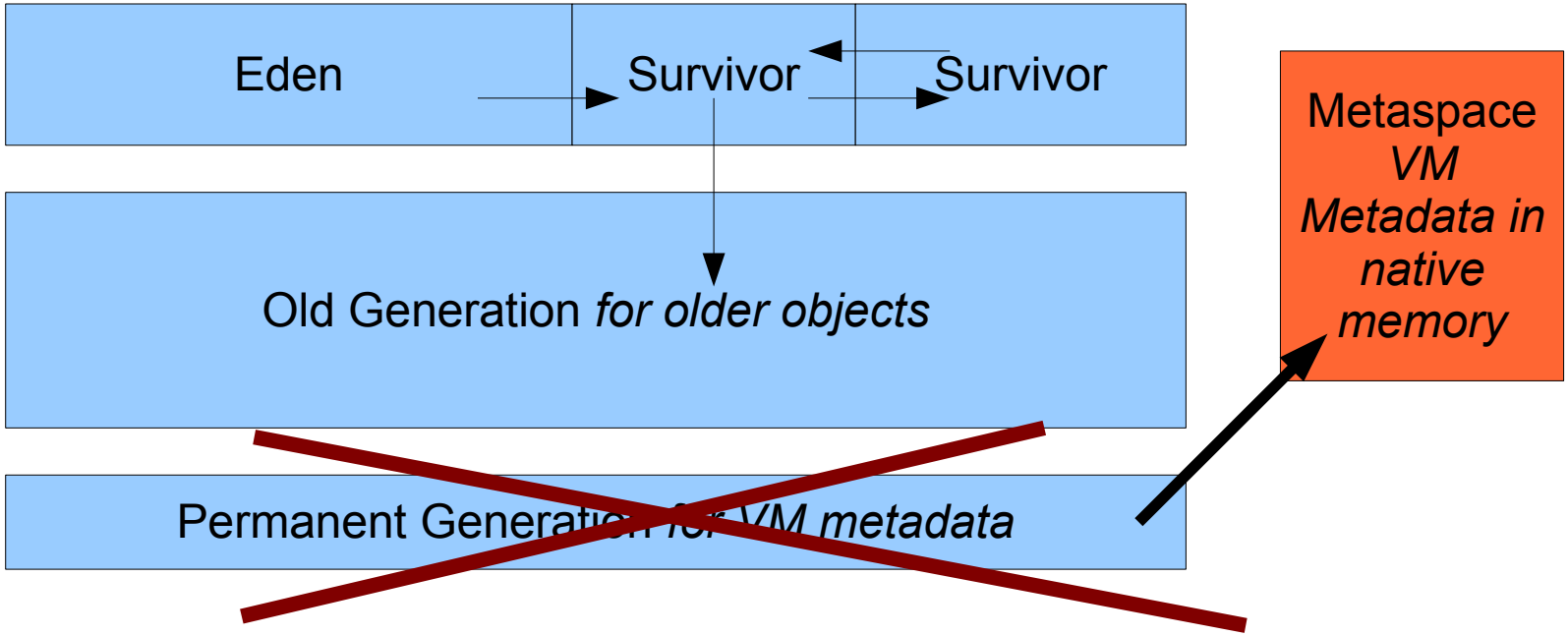
What was “PermGen”?

- “Permanent” Generation
- Region of Java heap for JVM Class Metadata
- Representation of Java classes
  - Class hierarchy information, fields, names
  - Method compilation information and bytecodes
  - Vtables
  - Constant pool and symbolic resolution
  - Interned strings (moved out of PermGen in 7)

Java Memory Layout with PermGen



Where did JVM Metadata go?



PermGen size

- Limited to MaxPermSize – default ~64M - 85M
- **Contiguous** with Java Heap
  - Identifying young references from old gen and permgen would be more expensive and complicated with a non-contiguous heap – card table
- Once exhausted throws OutOfMemoryError “PermGen space”
  - Application could clear references to cause class unloading
  - Restart with larger MaxPermSize
- Hard to predict
  - Size needed depends on number of classes, size of methods, size of constant pools, etc

Why was PermGen Eliminated?

- Fixed size at startup – applications ran out
  - -XX:MaxPermSize=… was hard to size
- Improve GC performance
  - **Special iterators** for metadata
  - Deallocate class data **concurrently** and not during GC pause
- Enable future improvements
  - were limited by PermGen (e.g. G1 concurrent class unloading)

Improving GC Performance

- During full collection, metadata to metadata pointers are not scanned
  - A lot of complex code (particularly for CMS) for metadata scanning was removed
- Metaspace contains few pointers into the Java heap
  - Pointer to java/lang/Class instance in class metadata
  - A component java/lang/Class pointer in array class metadata
- No compaction costs for metadata

Metaspace

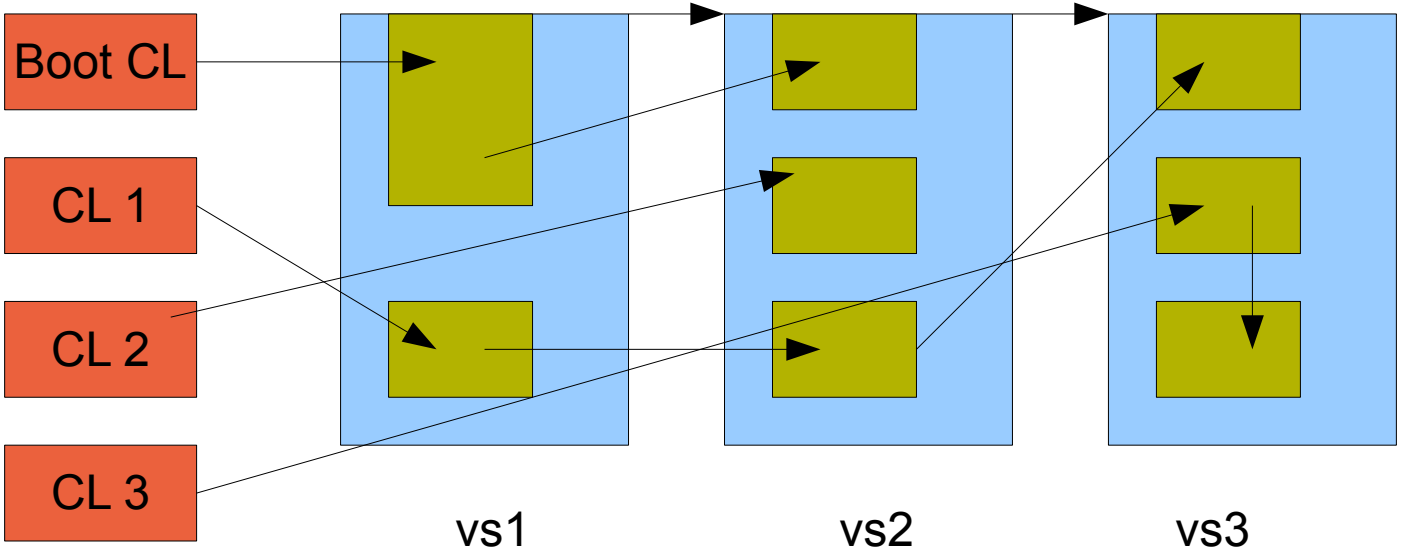
- Take advantage of Java Language Specification property
  - Class metadata lifetime same as their class loader's
- Per loader storage area – Metaspace (collectively called Metaspace)
  - **Linear** (bump) **allocation** only
  - **No individual reclamation** (except for RedefineClasses and class loading failure)
  - **Not scanned** by GC and **not compacted**
  - Metaspace-allocated objects **never relocate**
  - **Reclamation *en-masse*** when class loader found dead by GC

Metaspace Allocation

- **Multiple** mmap/VirtualAlloc virtual memory spaces
- Allocate **per-class loader** chunk lists
  - Chunk sizes depend on type of class loader
  - Smaller chunks for sun/reflect/DelegatingClassLoader, JSR292 anonymous classes
- Return chunks to free chunk lists
- Virtual memory spaces **returned** when emptied
- Strategies to **minimize fragmentation**

Metaspace Allocation

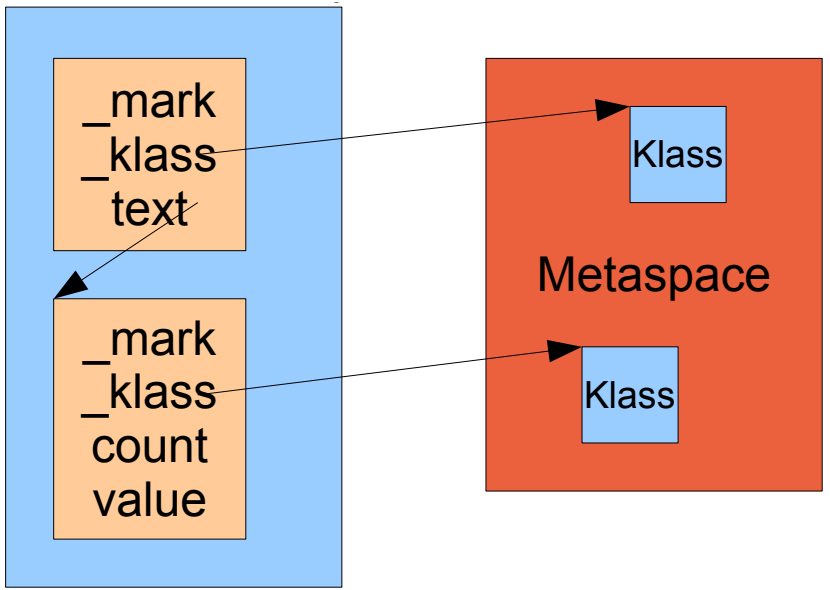
- Metachunks in virtual spaces (vs1, vs2, vs3...)



Java Object Memory Layout

```
class Message {
    String text;
    void add(String s) { ...}
    …
}
```

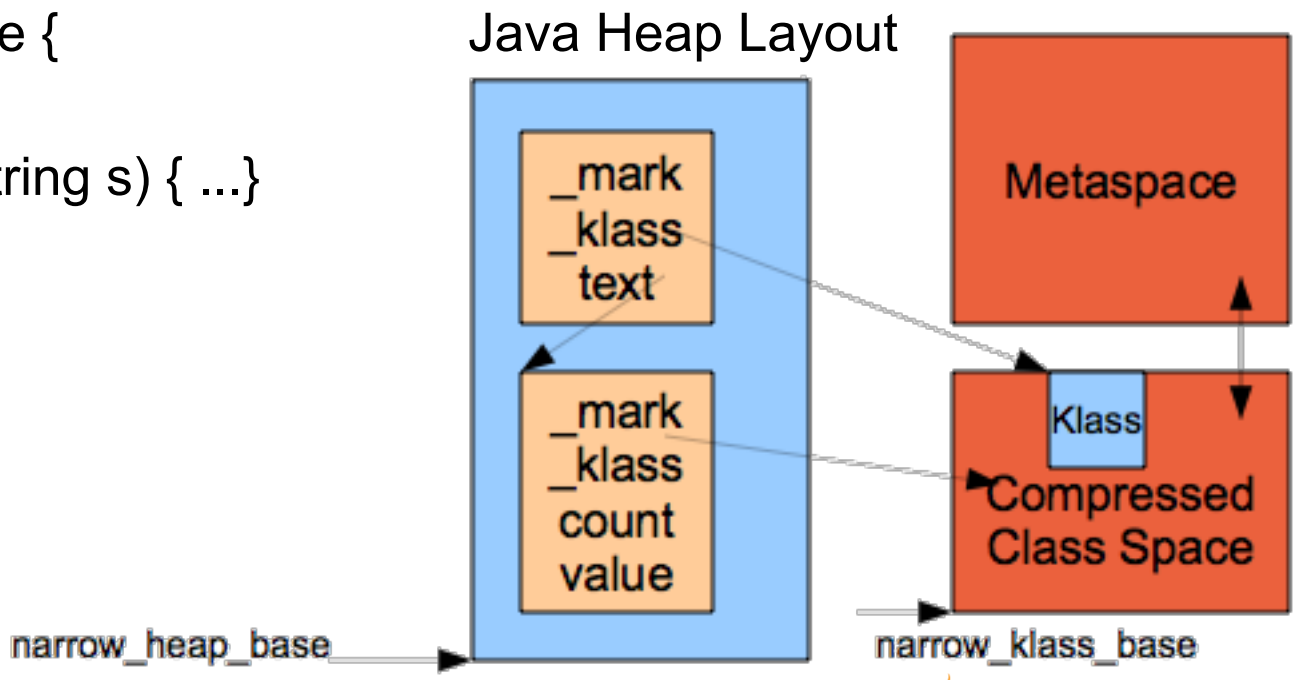
Java Heap Layout



Java Object Memory Layout

with compressed class pointers

```
class Message {
    String text;
    void add(String s) { ...}
    …
}
```



How to tune Metaspace?

- -XX:MaxMetaspaceSize={unlimited}

- Limit the memory used by class metadata before excess swapping and native allocation failure occurs
  - Use if suspected class loader memory leaks
  - Use if on 32-bit

How to tune Metaspace?

- -XX:MetaspaceSize={21M}

- Set to a higher limit if application loads more
- Possibly use same value set by PermSize to delay initial GC
- High water mark increases with subsequent collections for a reasonable amount of head room before next Metaspace GC

How to tune Metaspace?

- -XX:CompressedClassSpaceSize={1G}

- Only valid if -XX:+UseCompressedClassPointers (default on 64 bit)
- Not committed until used

Metaspace Monitoring and Management

- MemoryManagerMXBean with name MetaspaceManager
- MemoryPoolMXBeans
  - “Metaspace”
  - “Compressed Class Space”
- $ jmap -clstats <pid>
- $ jstat -gc <pid>
- $ jcmd <pid> GC.class_stats
- Java Mission Control (bundled with Oracle JDK 8)

PermGen removal

Summary

- Hotspot metadata is now allocated in Metaspace
  - Chunks in mmap spaces based on liveness of class loader
- Compressed class pointer space is still fixed size but large
- Tuning flags available but not required
- Change enables other optimizations and features in the future
  - Application class data sharing
  - Young collection optimizations, G1 class unloading
  - Metadata size reductions and internal JVM footprint projects

Nashorn

JavaScript engine for Java Platform

Nashorn

JVM support for dynamic languages

- Nashorn is written completely in Java
- Extensively uses JSR292
- Required numerous performance improvements
  - LambdaForms (JEP 160)
  - incremental inlining
  - exact math intrinsics
    - Math.addExact, Math.substractExact, etc
  - improved type profiling & type speculation

JVM support for dynamic languages

Exact Math intrinsics

```
public static long addExact(long x, long y) {
        long r = x + y;
        if (((x ^ r) & (y ^ r)) < 0) {
            throw new ArithmeticException("long overflow");
        }
        return r;
}
```

JVM support for dynamic languages

Exact Math intrinsics

Math.addExact(I1, Integer.MAX_VALUE)

Compiled version:

```
0x0...5d: add    $0x7fffffff,%r8
0x0...64: mov    %r8,%r9
0x0...67: xor    %r11,%r9
0x0...6a: mov    %r8,%r11
0x0...6d: xor    $0x7fffffff,%r11
0x0...74: and    %r11,%r9
0x0...77: test   %r9,%r9
0x0...7a: jl     0x0000000102c70e95 // slow path: throw exception
```

JVM support for dynamic languages

Exact Math intrinsics

Math.addExact(I1, Integer.MAX_VALUE)

Intrinsified version:

0x…1d: **add** $0x7fffffff,%rax

0x…24: **jo** 0x000000010c044b3f // **slow path**: overflow

Smaller features

- **JSR 308**: Annotations on Java Types
  - new class file attributes
- **JEP 171**: Fence Intrinsics
  - sun.misc.Unsafe: loadFence, storeFence & fullFence
- **JEP 136**: Enhanced Verification Errors
  - VerifyError with detailed error message
- **JEP 142**: Reduce Cache Contention on Specified Fields
  - @Contended

JSR 308: Annotations on Java Types

```
Map<@Interned Key, @NonNull Value> = new HashMap<>();
```

JSR 308: Annotations on Java Types

Class File Attributes

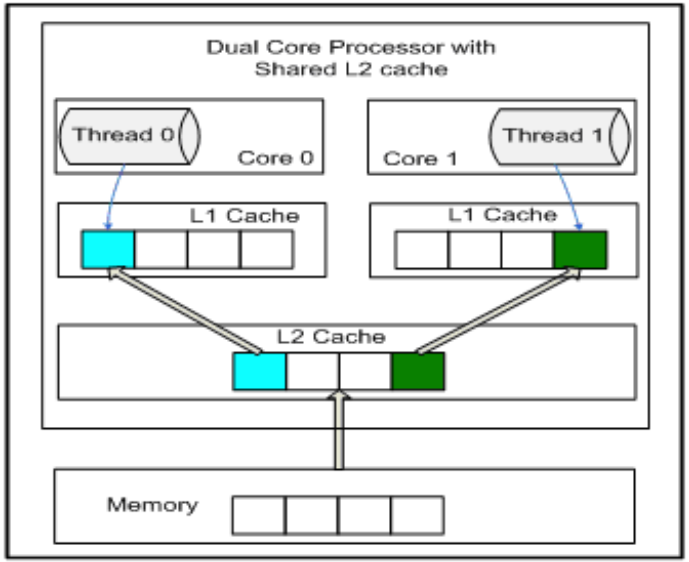
- New attributes:
  - RuntimeVisibleTypeAnnotations
  - RuntimeInvisibleTypeAnnotations
- Stored on the smallest enclosing class, field, method, or Code
- How to access:
  - javax.lang.model
  - javax.ide
  - com.sun.source.tree

JEP 171: Fence Intrinsics

- How to **express** memory model:
  - happens-before relations
  - memory barriers/fences
- Happens-before in Java Memory Model (JMM)
- sun.misc.**Unsafe**.[load|store|full]Fence introduces memory barriers
- Why:
  - some relations aren't possible to express in JMM at the moment
    - consider StoreStore

JEP 142: Reduce Cache Contention on …

- What is **cache contention** and **false sharing**?
  - adjacent memory accesses can produce unnecessary memory traffic on SMP systems



JEP 142: Reduce Cache Contention on …

- How to avoid it?
  - manually “pad” contended fields

```
public class A {
    int x;
    int i01, i02, …, i16;
    int y;
    int i17, i18, …, i32;
}
```

JEP 142: Reduce Cache Contention on …

- How to avoid it?
  - mark them @Contended

```
public class A {
    int x;
    @Contended int y;
}
```

JEP 142: Reduce Cache Contention on …

- **sun.misc**.Contended
- How to use: -XX:-RestrictContended
  - by default, has effect only in privileged context (on boot class path)

JEP 136: Enhanced Verification Errors

Before

```
Exception in thread "main" java.lang.VerifyError:
    Bad instruction in method Test.main([Ljava/lang/String;)V at offset 0
```

JEP 136: Enhanced Verification Errors

Now

```
Exception in thread "main" java.lang.VerifyError: Bad instruction
Exception Details:
  Location:
    Test.main([Ljava/lang/String;)V @0: <illegal>
  Reason:
    Error exists in the bytecode
  Bytecode:
    0000000: ff00 0200 004c 2b04 b800 03b9 0004 0200
    0000010: 57b1
```

MAKE THE FUTURE JAVA

ORACLE