Java in a World of Containers

mikael.vidstedt@oracle.com
Not-coder, JVM
@MikaelVidstedt

matthew.gilliard@oracle.com
Coder, not-JVM
@MaximumGilliard

JavaYourNext
(Cloud)

FnProject

- http://fnproject.io
- Open Source Functions-as-a-Service
- “functions” packaged as container images

- Cold Latency MATTERS!

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda

Java in a World of Containers

Creating Docker images

Creating Custom JREs

Java + Docker features

Java in a World of Containers

In a World of Containers We Expect…

- Safety and security becoming increasingly more important
- Sprawl
  - Many instances
  - Mix of different applications
  - Heterogeneous machines
  - Heterogeneous container configurations

https://citelighter-cards.s3.amazonaws.com/p17k6i1bqs3okg8gnisklkg1k0_51924.png

Java in a World of Containers

Java's characteristics make it ideal for a container environment

- Managed language/runtime
- Hardware and operating system agnostic
- Safety and security enforced by JVM
- Reliable: Compatibility is a key design goal
- Runtime adaptive: JVM ensures stable execution when environment changes
- Rich eco system

Java in a World of Containers

Java's characteristics make it ideal for a container environment

- Managed language/runtime
- Hardware and operating system agnostic
- Safety and security enforced by JVM
- Reliable: Compatibility is a key design goal
- Runtime adaptive: JVM ensures stable execution when environment changes
- Rich eco system
- **We are committed to keeping Java the first choice for container deployments**

Creating Docker images

Docker

```
<dir>/
      Dockerfile
      jdk-9+181_linux-x64_bin.tar.gz
      HelloWorld.class
```

Dockerfile

```
FROM oraclelinux:7-slim
ADD jdk-9+181_linux-x64_bin.tar.gz /opt/jdk
ENV PATH=$PATH:/opt/jdk/jdk-9/bin
ADD HelloWorld.class /
CMD [ "java", “-showversion”, ”HelloWorld" ]
```

Docker

```
# docker build -t my/jdk9 .
Sending build context to Docker daemon 346.3 MB
Step 1/4 : FROM oraclelinux:7-slim
. . .
Successfully built df05dbf52403

# docker run --rm my/jdk9
java version "9”
Java(TM) SE Runtime Environment (build 9+181)
Java HotSpot(TM) 64-Bit Server VM (build 9+181, mixed mode)
Hello, world!
```

Creating Custom JREs

Custom JREs

- A Docker image containing the full JDK is large (500+ MB)
  - Contains the stuff you want: `java.{lang,util,…}.*, javax.management.*, …`
  - And all the stuff you don't: `corba, jaxws, …`
- JDK 9 introduces a module system and tooling for creating custom JREs
  - Only include the modules/functionality needed for the application
  - A minimal JRE only includes the `java.base` module
    - Can be enough for many applications
  - jdeps can help identify which modules an application uses

```
# $JAVA_HOME/bin/jdeps lib/tomcat-api.jar
tomcat-api.jar -> java.base
tomcat-api.jar -> java.instrument
tomcat-api.jar -> java.naming
...
```

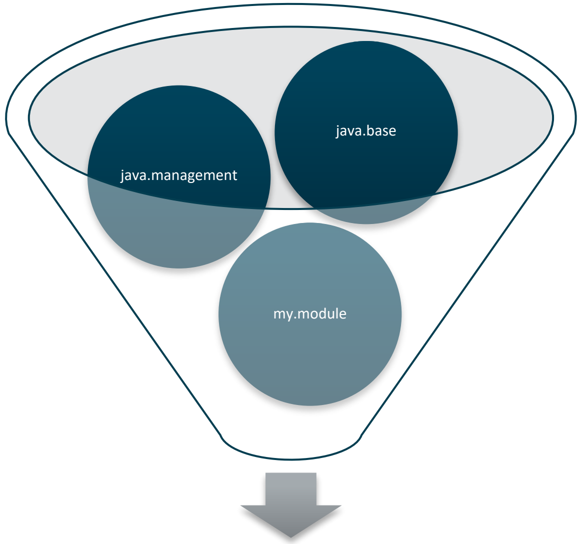
Creating a Custom JRE

- Creating a custom JRE is straightforward

  -
    ```
    jlink <options>
       --module-path <modulepath>
       --add-modules <module>[,<module>...]
    ```
- Example: Creating a java.base (only) JRE

  -
    ```
    $JAVA_HOME/bin/jlink
       --output my-jre
       --module-path $JAVA_HOME/jmods
       --add-modules java.base
    ```

  -
    ```
    my-jre/bin/java HelloWorld
    ```
- Note: The application does not have to be module aware!



Custom JRE

Multi-Stage Dockerfile

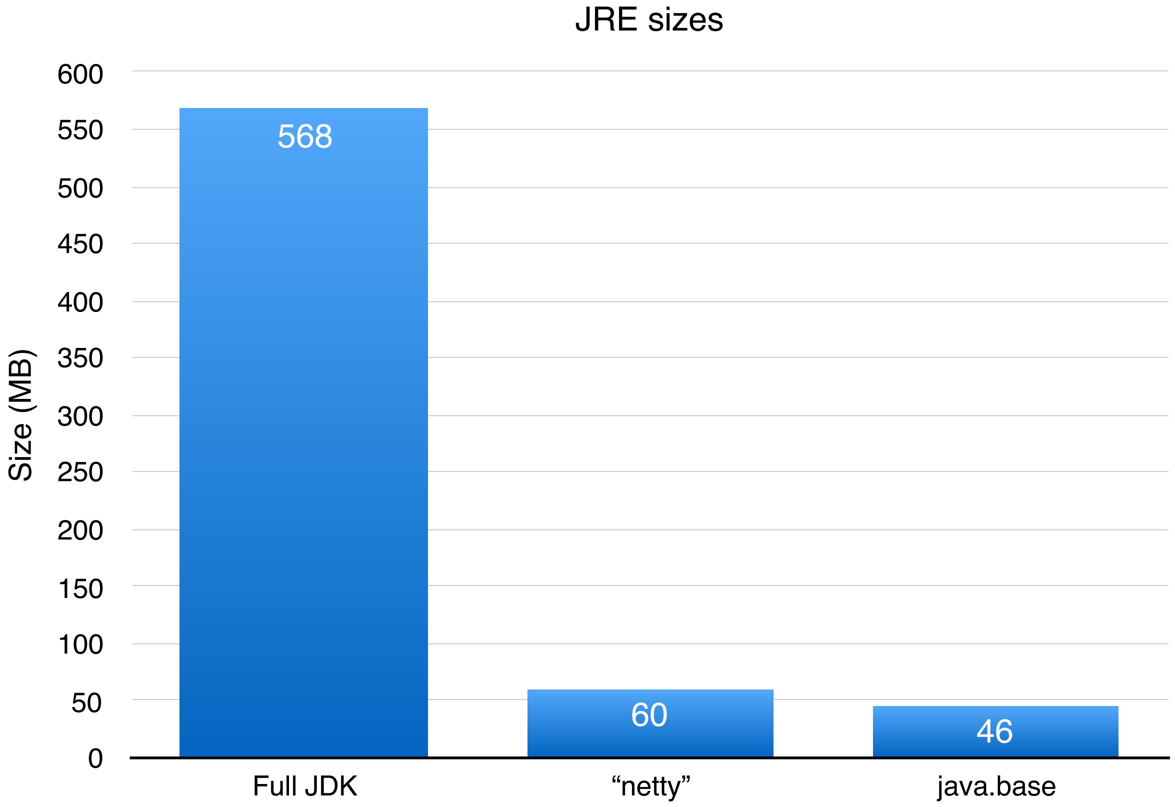
- Creation of custom JRE Docker image can be automated using Multi-Stage Dockerfiles

```
# Multi-Stage example using jlink to produce small Java runtime
FROM openjdk:9-jdk AS java-build
WORKDIR /jlink/outputdir
RUN jlink --module-path /docker-java-home/jmods --strip-debug --
compress=2 --output java --add-modules java.base

FROM alpine:latest
WORKDIR /root/
COPY --from=java-build /jlink/outputdir .
ENV PATH /root/java/bin:$PATH
CMD ["bash”]
```

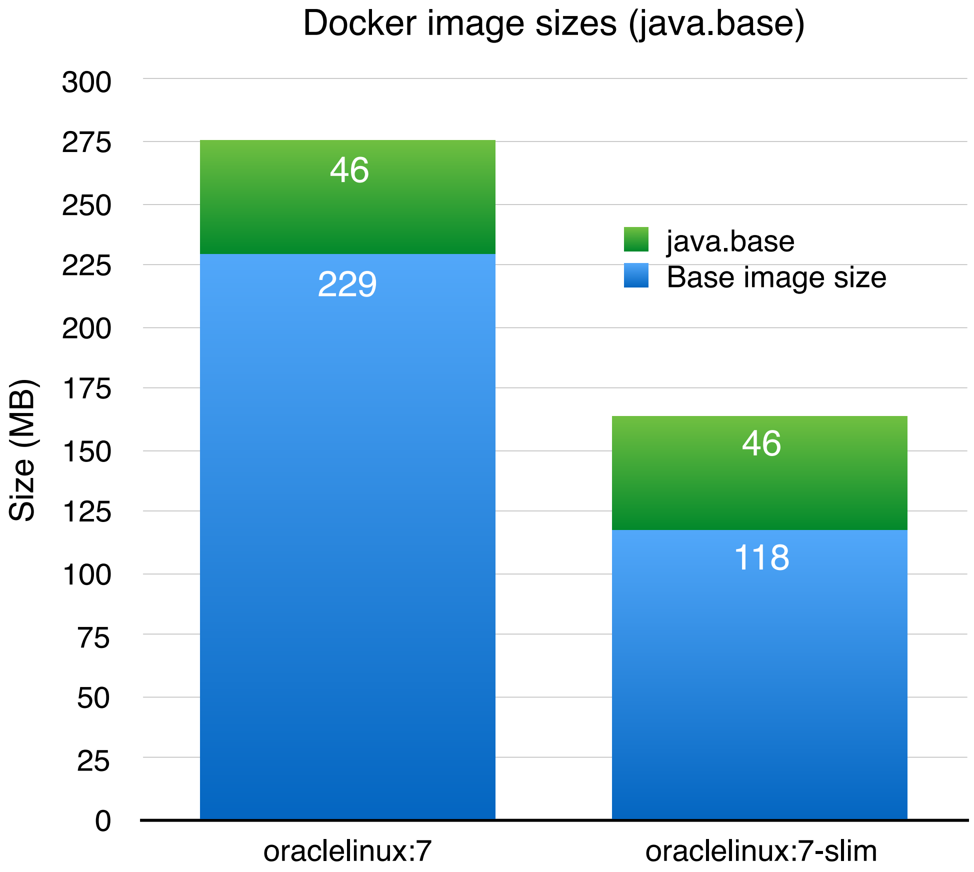
Optimizing the Size of the JRE

- Full JDK: Default JDK (not jlink:ed)
- java.base: `jlink --add-modules java.base`
- "netty": A set of modules expected to be sufficient for many/most Java applications
  - `java.base, java.logging, java.management, java.xml, jdk.management, jdk.unsupported`
  - Note: Does **not** include the netty application itself
- Size can be further optimized
  - jlink `--compress` can reduce size by 25%+



Optimizing the Base Image Size

- Base image is a significant part of the total image size
- With Docker the exact base image matters less
  - As long as it can still run Java



Alpine Linux & musl libc

Small. Simple. Secure.

Alpine Linux is a security-oriented, lightweight Linux distribution based on **musl libc** and busybox.

- https://www.alpinelinux.org

musl is lightweight, fast, simple, free, and strives to be correct in the sense of standards-conformance and safety.

- https://www.musl-libc.org

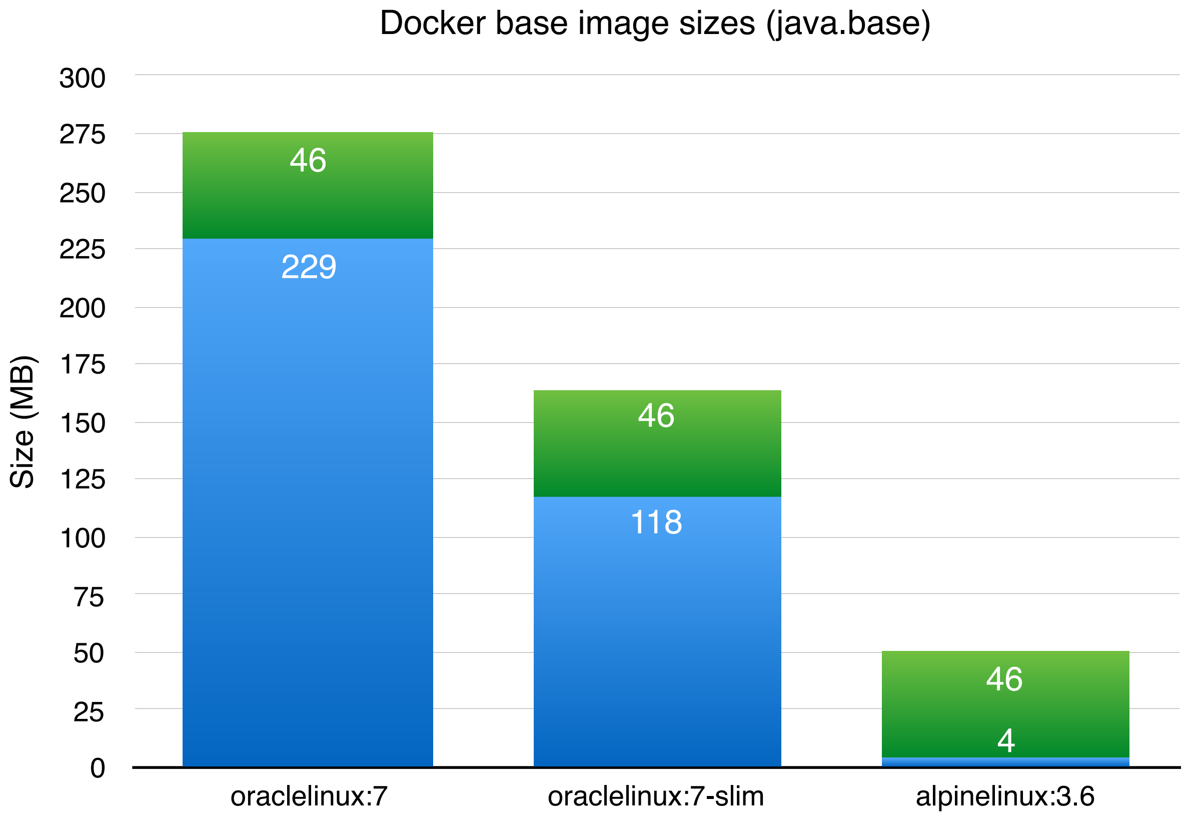
OpenJDK Project “Portola”

- OpenJDK project “Portola” provides a port of the JDK to Alpine/musl
- The Alpine Linux base image weighs in at **4MB**
  - Uses the “musl” C library

http://openjdk.java.net/projects/portola/

http://hg.openjdk.java.net/portola/portola

Portola-dev@openjdk.java.net



OpenJDK Project “Portola”

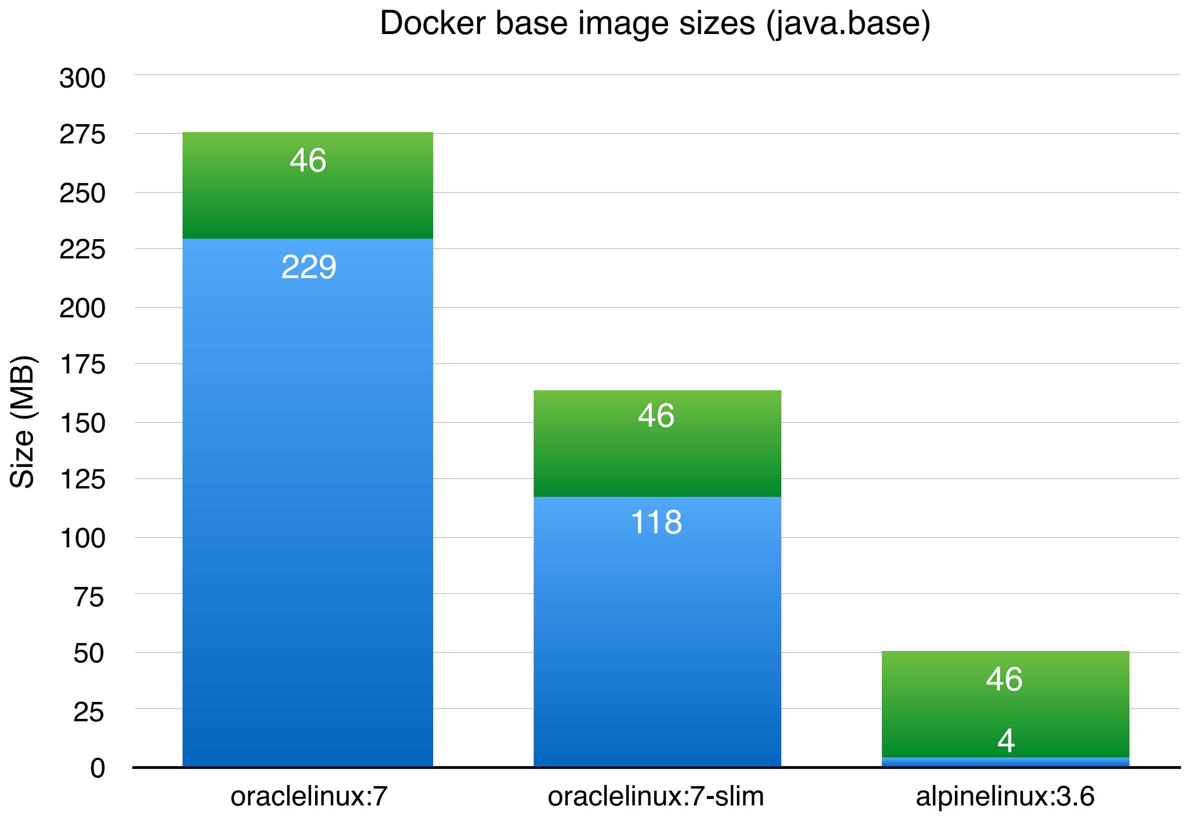
- OpenJDK project “Portola” provides a port of the JDK to Alpine/musl
- The Alpine Linux base image weighs in at **4MB**
  - Uses the “musl” C library

http://openjdk.java.net/projects/portola/

http://hg.openjdk.java.net/portola/portola

Portola-dev@openjdk.java.net

Any interest in an Alpine port?



OpenJDK Project "Portola"

- OpenJDK project "Portola" provides a port of the JDK to Alpine/musl
- The Alpine Linux base image weighs in at **4MB**
  - Uses the "musl" C library

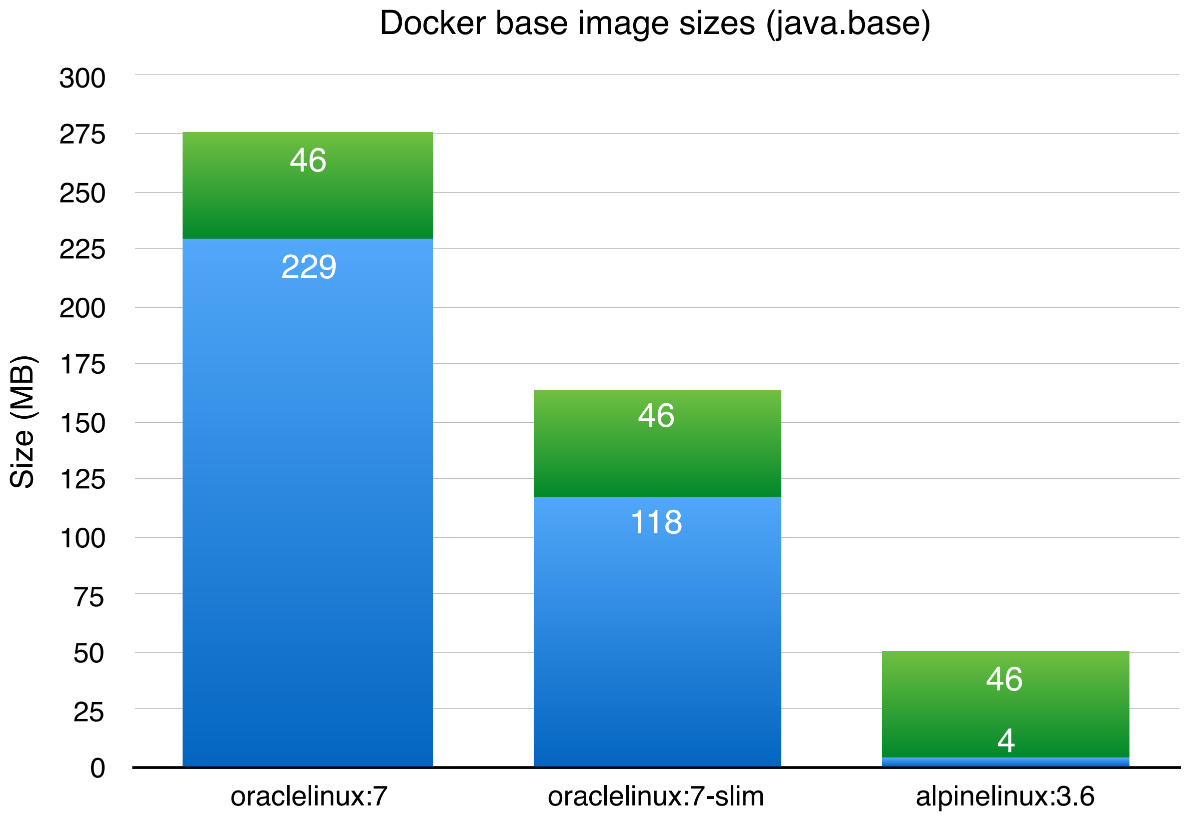
http://openjdk.java.net/projects/portola/

http://hg.openjdk.java.net/portola/portola

Portola-dev@openjdk.java.net

Any interest in an Alpine port?

Any interest in helping maintain it?

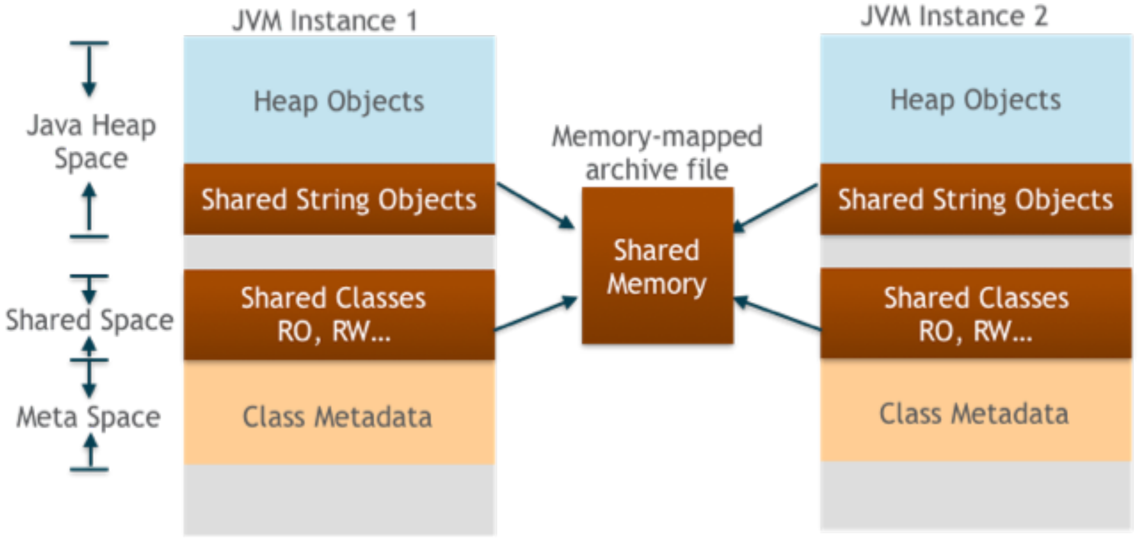


Java + Docker features

Sharing Across Instances

- Micro-services and Docker encourages running **many processes** on the same machine
- Chances are many instances will be running the **exact same application**
- OS shared libraries allows for sharing native data
- libc, libjvm.so all get shared automatically by the OS & Docker
  - Assuming same layer/file/inode
- What about Java class data?

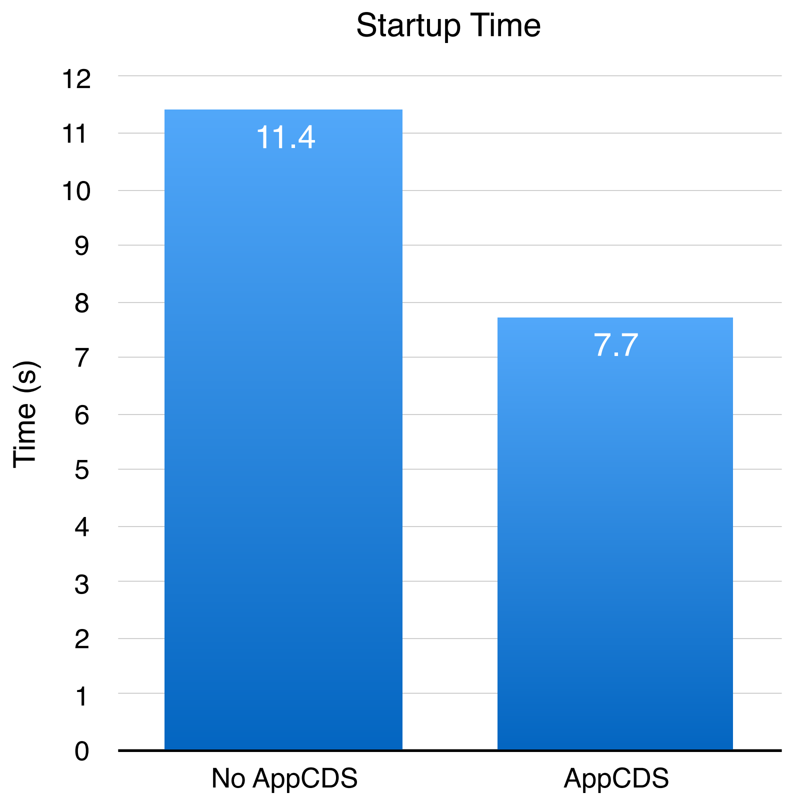
Class Data Sharing (CDS)

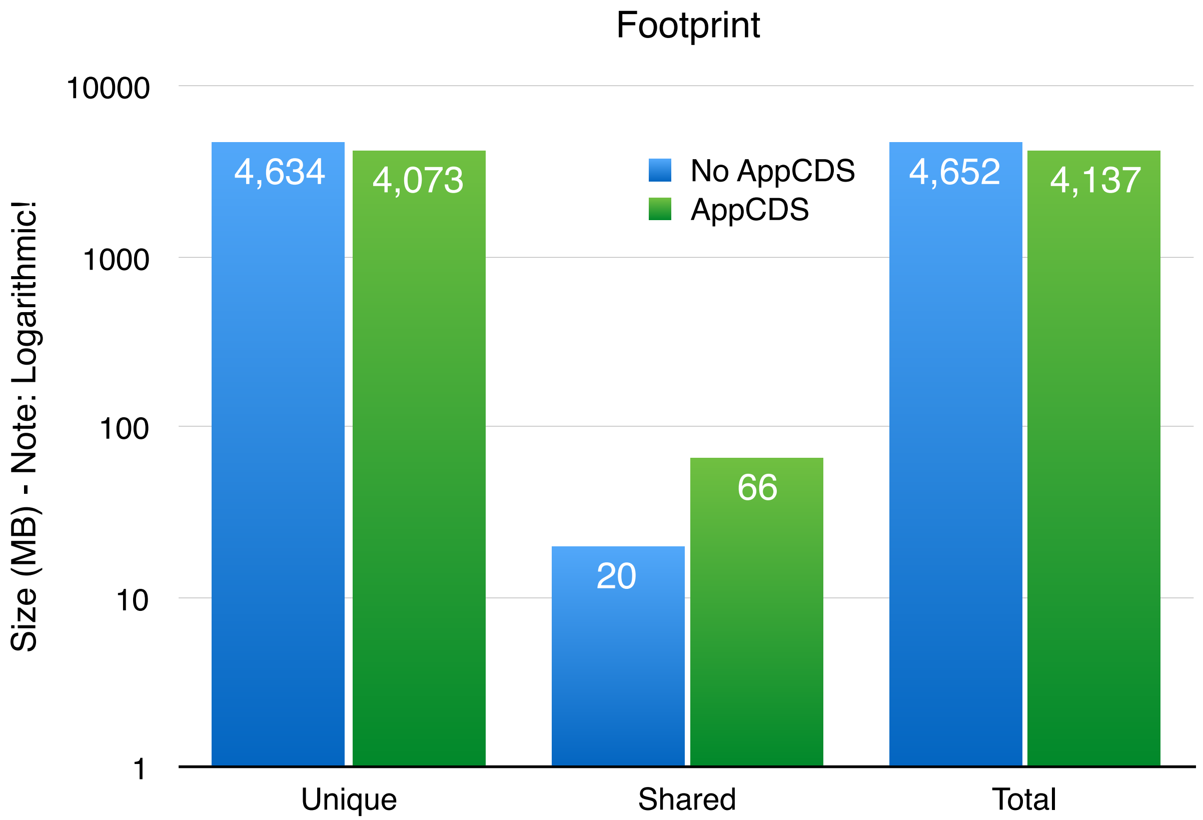


- Like OS shared libraries for Java class data
- Archive is memory-mapped
- RO pages shared, RW pages are shared copy-on-write
- Classes read from mapped memory without overhead of searching, reading & parsing from JAR files
- Archive can be shared across Docker containers

AppCDS Benefits - Startup Time and Footprint

Example: WebLogic Server Base Domain





- Sharing & savings increases with every instance
- With 10 instances there is ~10% saving in total memory footprint
- Can be good to use separate layers

Experimental: Ahead-of-Time Compilation (AOT)

- Like AppCDS, but for JIT compiled code
  - Pre-compiled code stored to archive
  - Allows sharing, footprint reduction, and reduced startup

Honoring Docker/cgroups Resource Limits

- The JVM has plenty of ergonomics which are based on the underlying system
  - Memory, #cpus, exact CPU model, etc.
  - For example, heap size is based on available memory
- Docker allows for specifying resource limits
  - Implemented through cgroups
  - **Not** transparent - requires cooperative application support
  - Explicit support needed for JVM

Honoring Docker/cgroups Resource Limits: CPU

- The JDK honors Docker CPU settings
  - `--cpuset-cpus` (JDK 9)
  - `--cpus, --cpu-shares, --cpu-quota` (JDK 10)
    - JDK-8146115: Improve docker container detection and resource configuration usage
- Reflected in
  - `Runtime.availableProcessors()`, ForkJoin pool, VM internal thread pools
  - Libraries/frameworks such as core.async, ElasticSearch, Netty

https://mjg123.github.io/2018/01/10/Java-in-containers-jdk10.html

Honoring Docker/cgroups Resource Limits: Memory

- Memory settings (JDK 10)
  - `-m<size>`
  - Reflected in
    - Java heap size, GC region sizes, other VM internal structures like code cache, …
- JDK-8186248: Allow more flexibility in selecting Heap % of available RAM

```
-XX:InitialRAMPercentage
-XX:MaxRAMPercentage
-XX:MinRAMPercentage
```

https://mjg123.github.io/2018/01/10/Java-in-containers-jdk10.html

Other

- JDK-8179498: attach in linux should be relative to /proc/pid/root and namespace aware (JDK 10)
- JDK-8193710: jcmd -l and jps commands do not list Java processes running in Docker containers (JDK 11)
- More to come
  - Draft JEP: JDK-8182070: Container aware Java

Java™

ORACLE®

BACKUP

jshell snippets (**snippets.txt**)

```
long javaMaxMem() {
    return Runtime.getRuntime().maxMemory();
}

import java.nio.file.*;

long sysMaxMem() throws IOException {
    return Files.lines(Paths.get("/sys/fs/cgroup/memory/memory.limit_in_bytes"))
        .mapToLong(Long::valueOf)
        .findFirst().getAsLong();
}
```

jshell in Docker execution commands

```
# Run without resource restrictions
docker run \
  --rm -it --volume $PWD:/in jdk-9-alpine \
  jshell /in/snippets.txt

# Run with resource restrictions but no Java configutation
docker run -m=384M --cpuset-cpus=0 \
  --rm -it --volume $PWD:/in jdk-9-alpine \
  jshell /in/snippets.txt

# Run with resource restrictions and Java configutation
docker run -m=384M --cpuset-cpus=0 \
  --rm -it --volume $PWD:/in jdk-9-alpine \
  jshell /in/snippets.txt \
  -J-XX:+UnlockExperimentalVMOptions -J-XX:+UseCGroupMemoryLimitForHeap \
  -R-XX:+UnlockExperimentalVMOptions -R-XX:+UseCGroupMemoryLimitForHeap
```