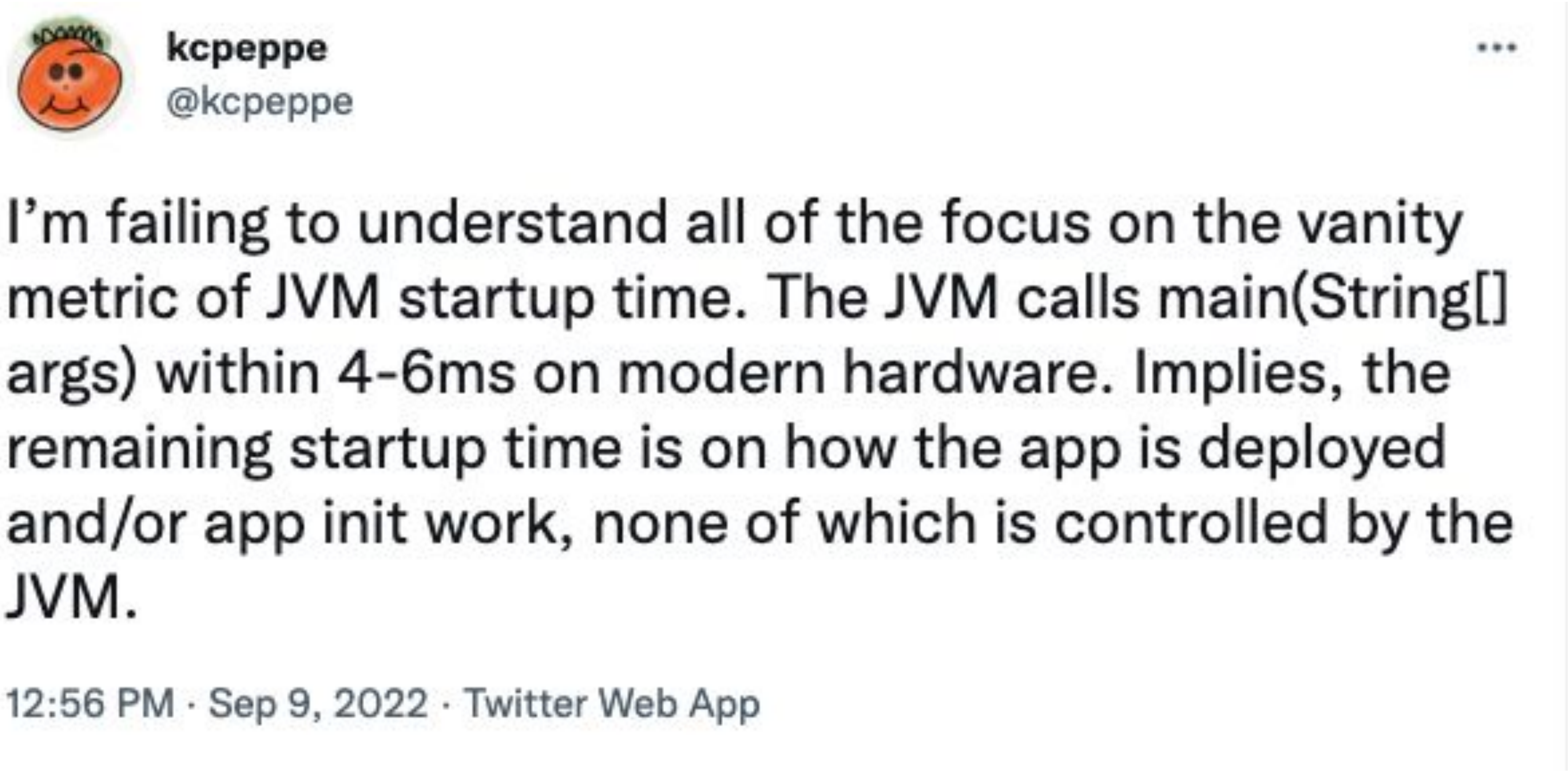
The Fast Startup Landscape is Expanding!

Dan Heidinga

Principal Software Engineer

@danheidinga

DanHeidinga

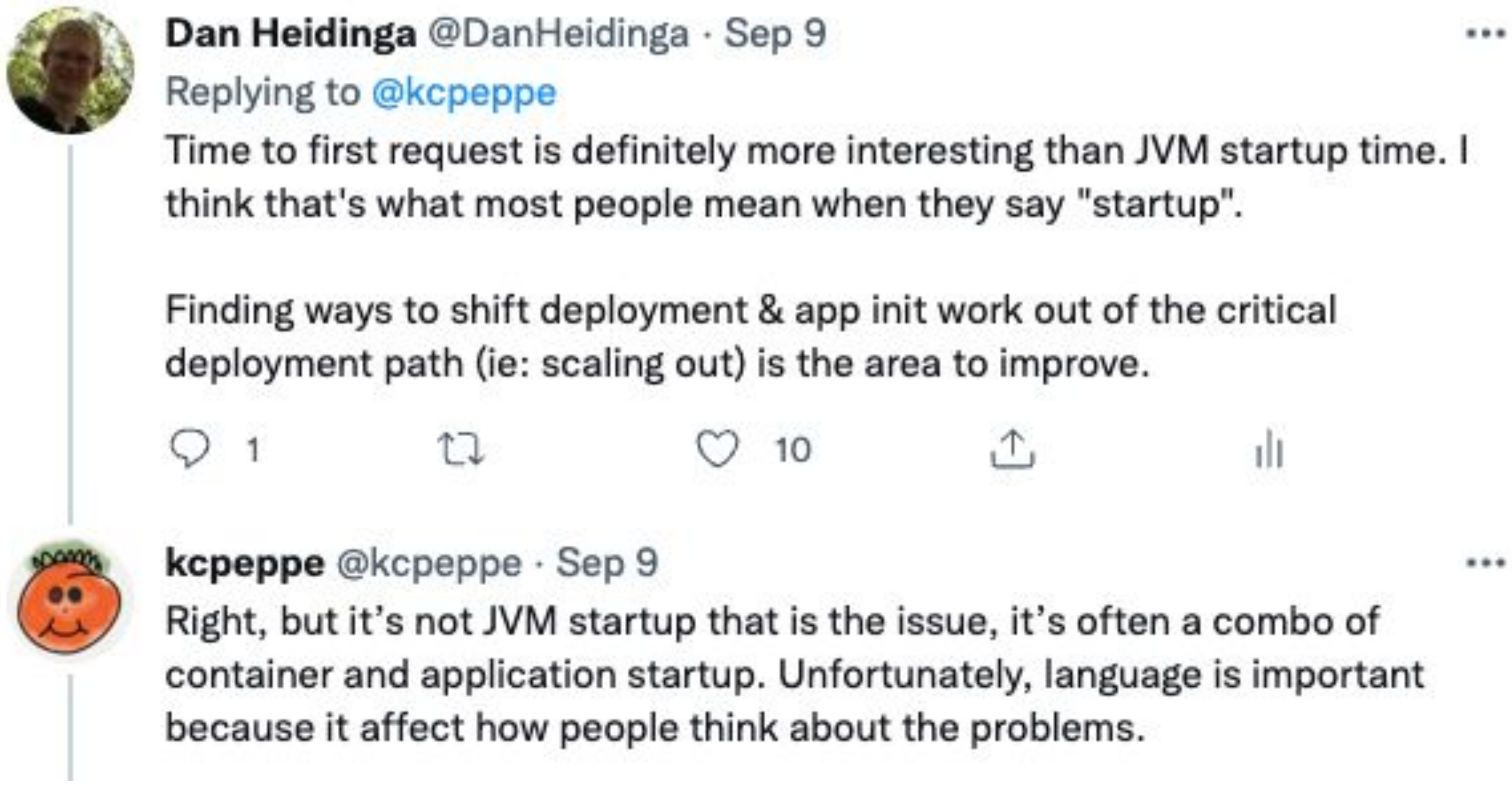


kcpeppe
@kcpeppe

I'm failing to understand all of the focus on the vanity metric of JVM startup time. The JVM calls main(String[] args) within 4-6ms on modern hardware. Implies, the remaining startup time is on how the app is deployed and/or app init work, none of which is controlled by the JVM.

12:56 PM · Sep 9, 2022 · Twitter Web App

https://twitter.com/kcpeppe/status/1568282140869275648



Dan Heidinga @DanHeidinga · Sep 9

Replying to @kcpeppe

Time to first request is definitely more interesting than JVM startup time. I think that's what most people mean when they say "startup".

Finding ways to shift deployment & app init work out of the critical deployment path (ie: scaling out) is the area to improve.

kcpeppe @kcpeppe · Sep 9

Right, but it's not JVM startup that is the issue, it's often a combo of container and application startup. Unfortunately, language is important because it affect how people think about the problems.

https://twitter.com/DanHeidinga/status/1568425183538470913

Why do we care about startup? More deployments!

Cloud

Microservices

Horizontal scaling

CI/CD

Frequent deployments

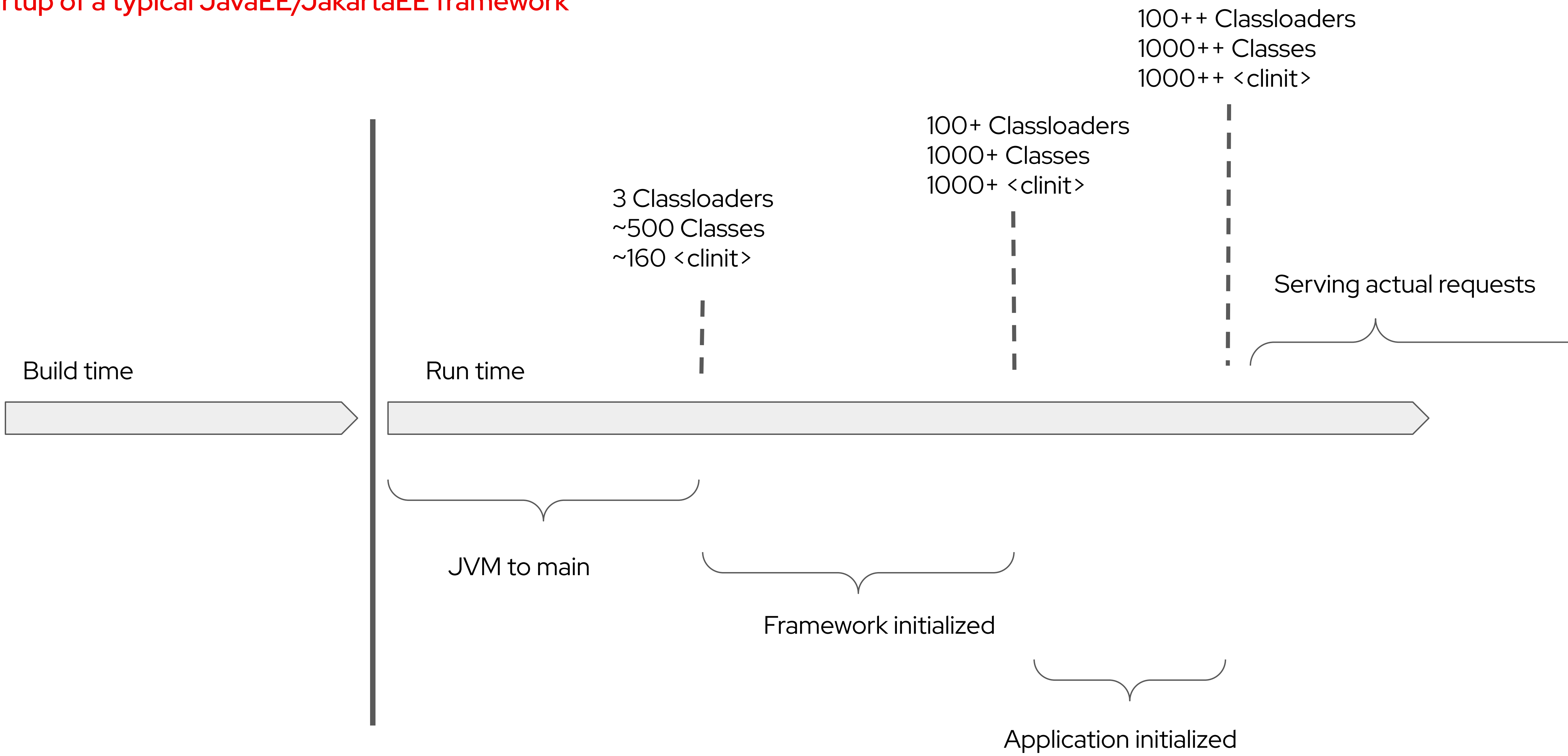
Serverless

cgi-bin model of deployment

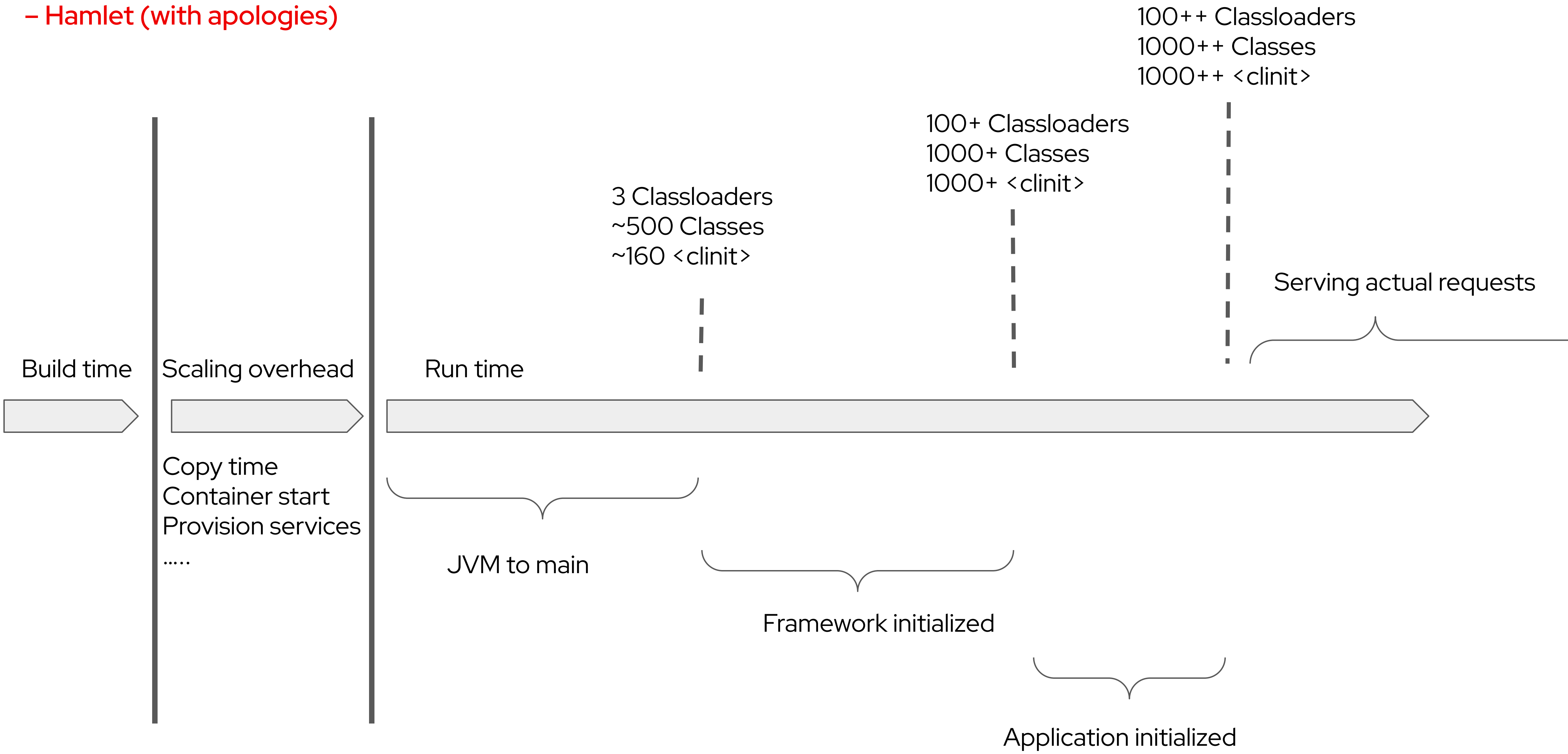
Scale to zero

RAM x CPU = $$

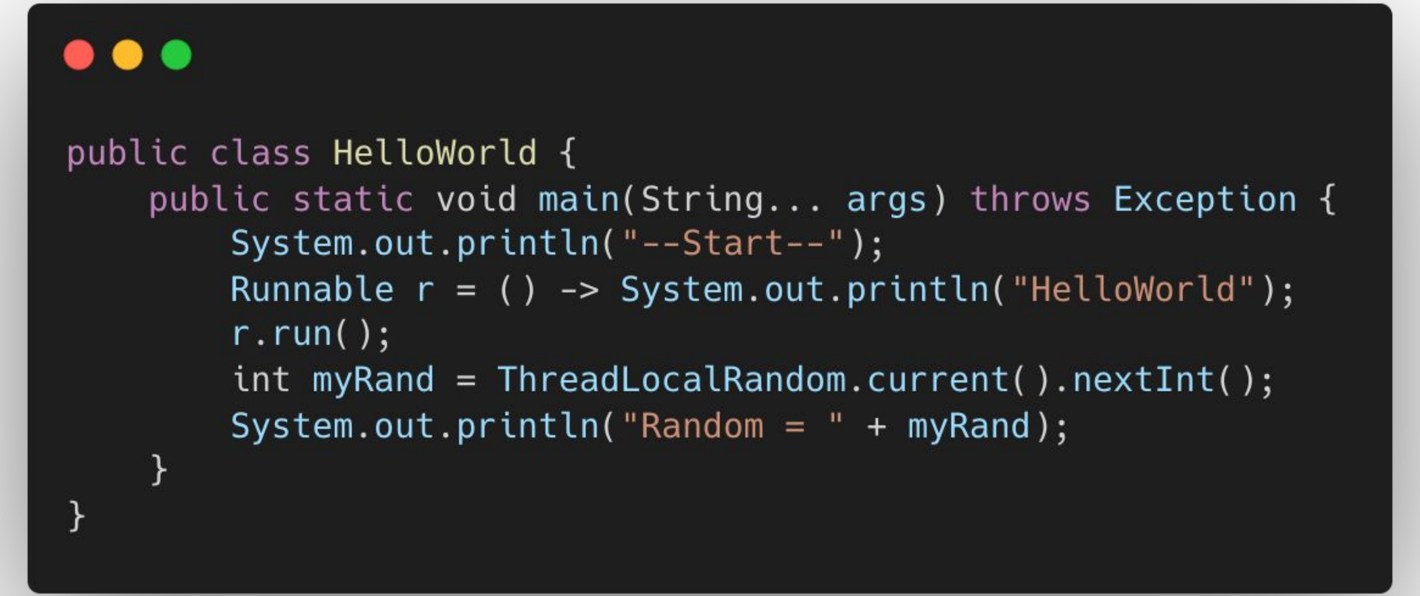
Startup of a typical JavaEE/JakartaEE framework



"There are more [sources of delay] than are dreamt of in your philosophy"
– Hamlet (with apologies)



Java: Dynamic Island

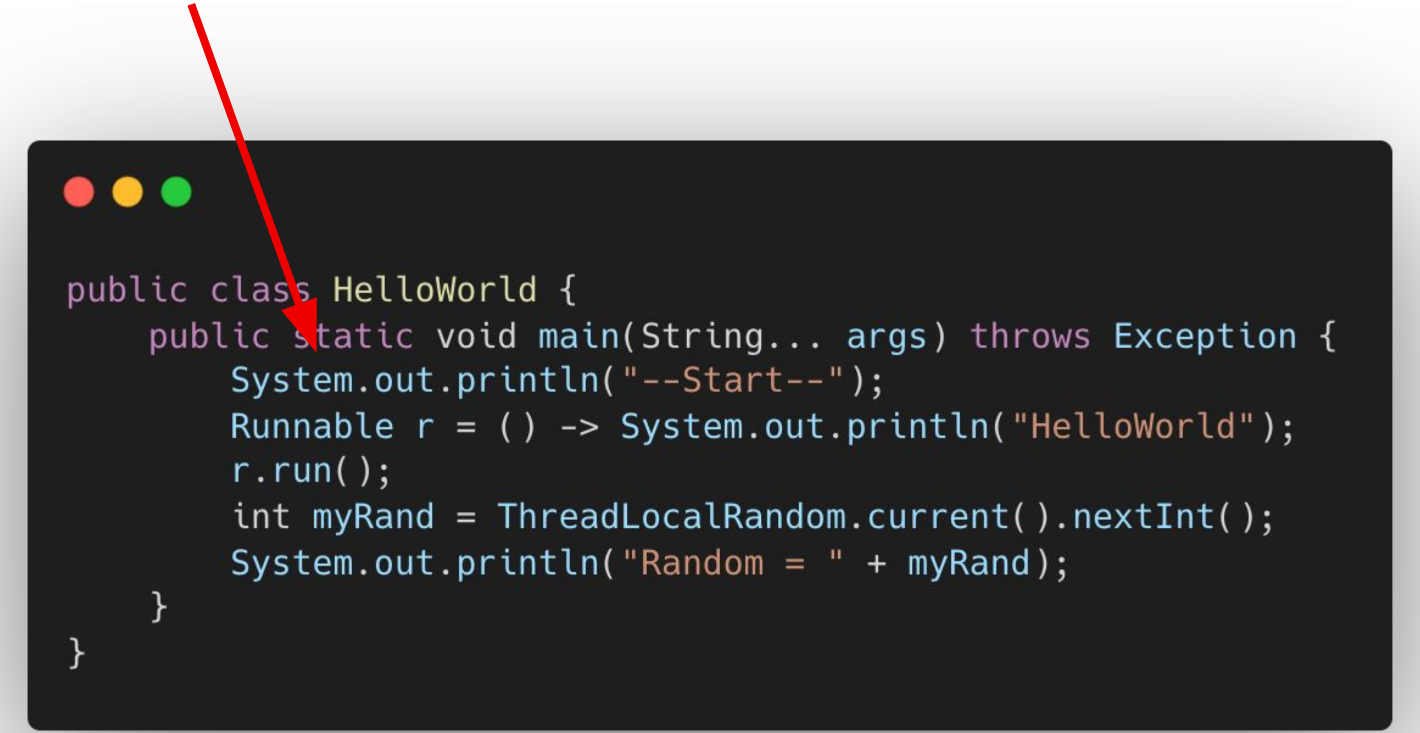


```java
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Dynamic Island

Dynamic classloading



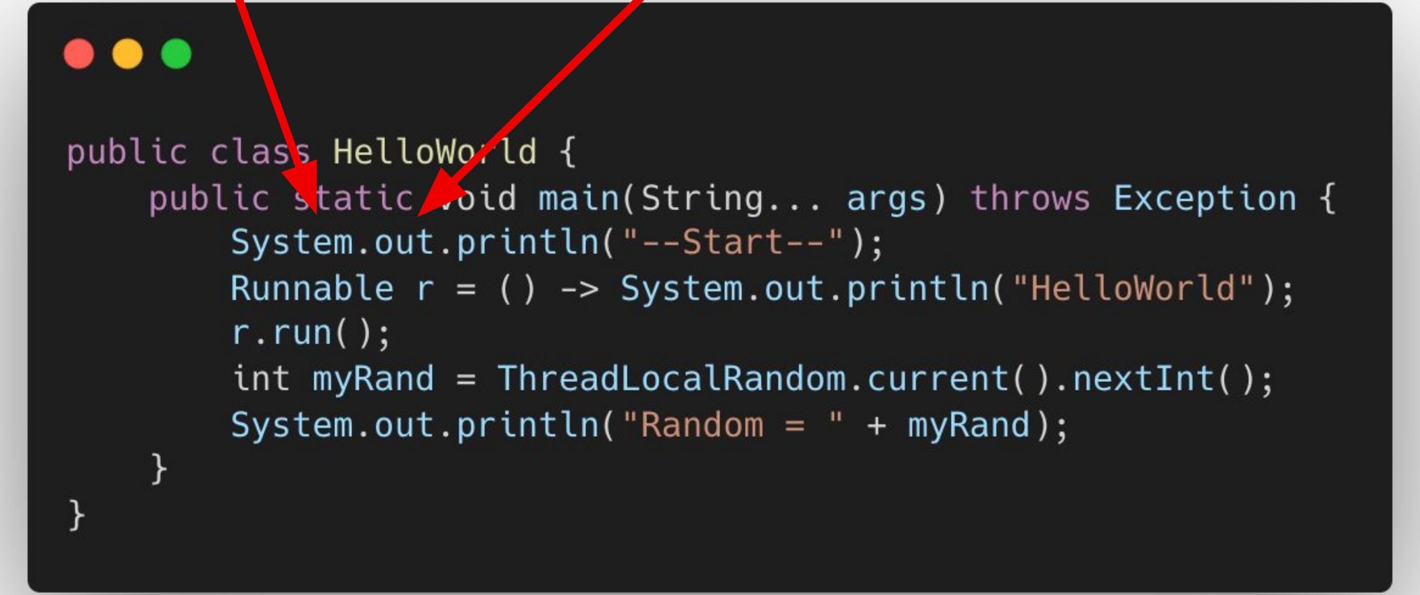
```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Dynamic Island

Dynamic classloading

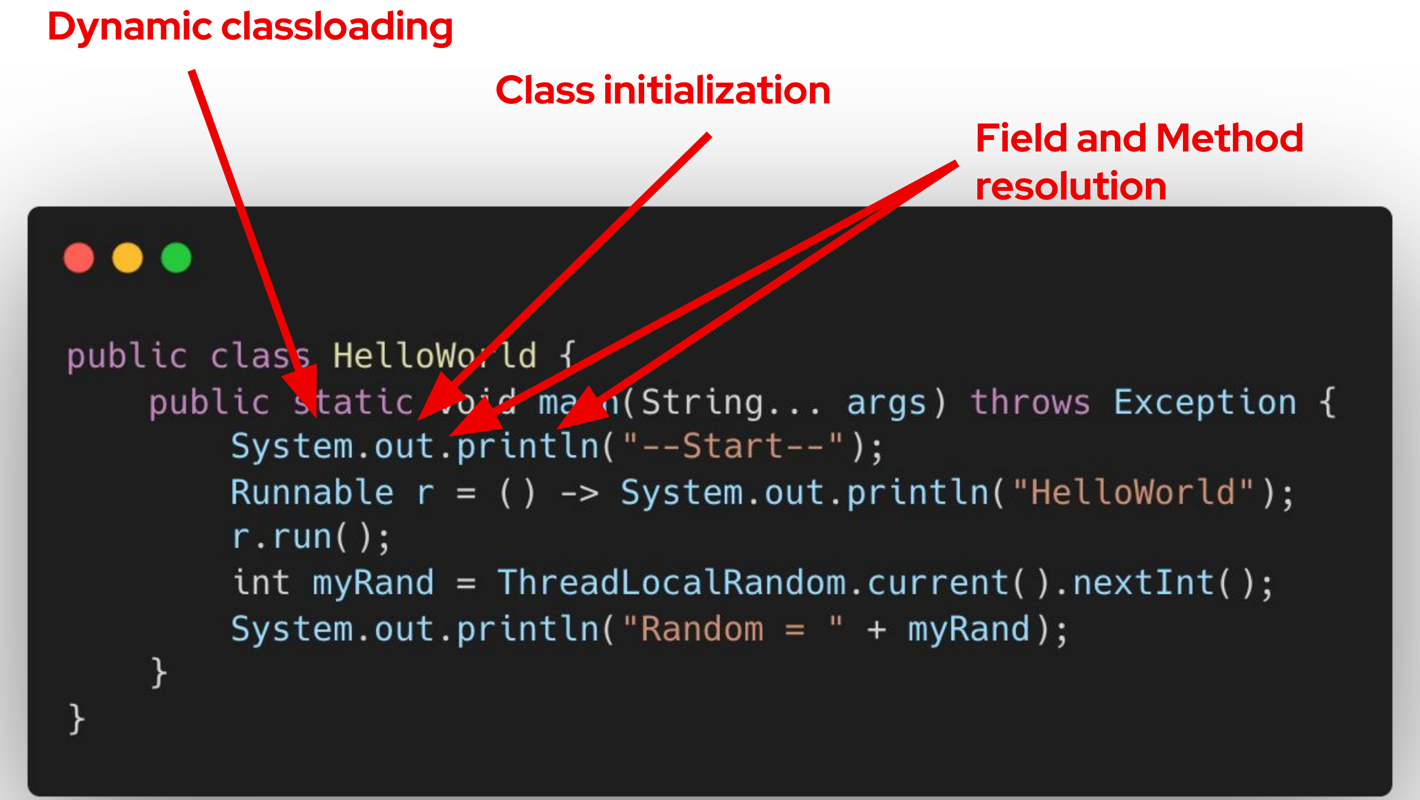
Class initialization



```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Dynamic Island



Dynamic classloading

Class initialization

Field and Method resolution

```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

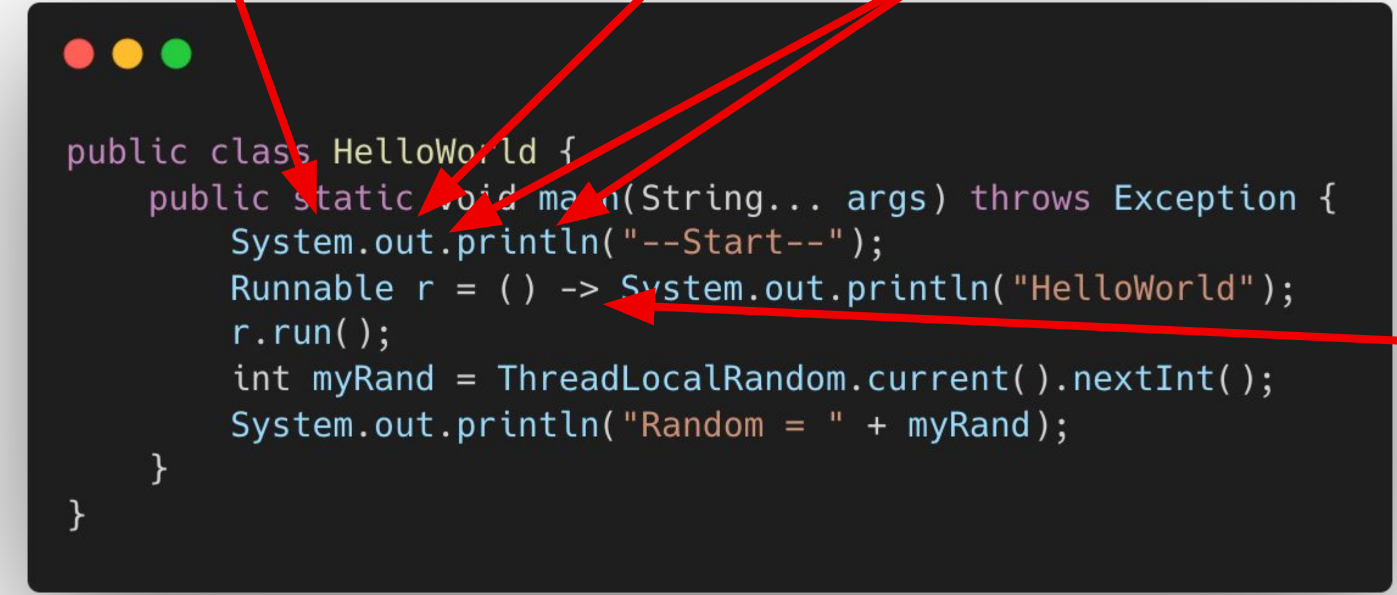
https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Dynamic Island

Dynamic classloading

Class initialization

Field and Method resolution



```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

MethodHandles
Reflection
Dynamic class generation

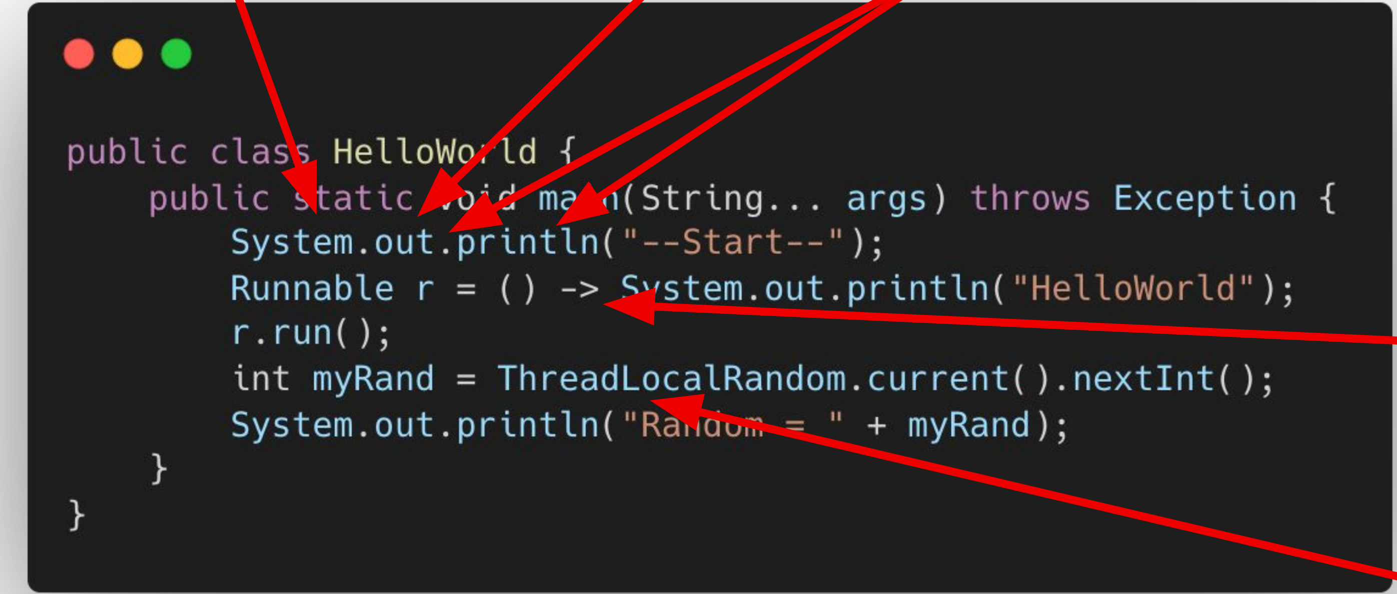
https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Dynamic Island

Dynamic classloading

Class initialization

Field and Method resolution



```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

MethodHandles
Reflection
Dynamic class generation

Dynamic classloading

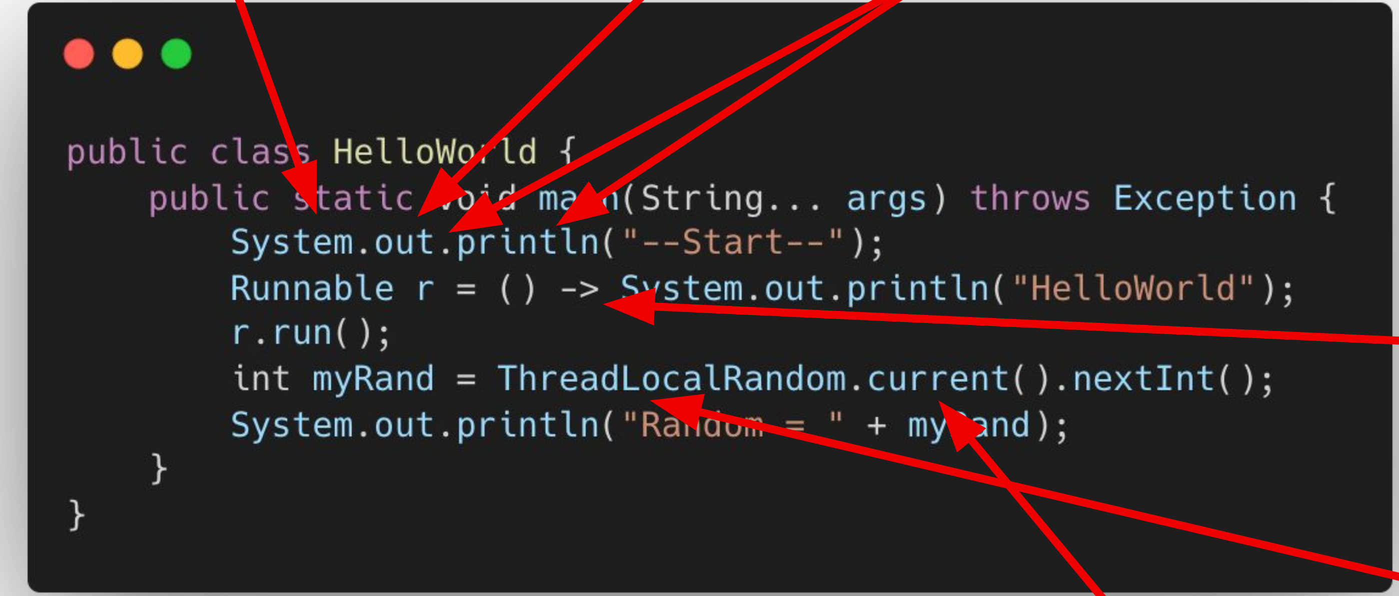
https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Dynamic Island

Dynamic classloading

Class initialization

Field and Method resolution



```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

MethodHandles
Reflection
Dynamic class generation

Dynamic classloading

Class initialization

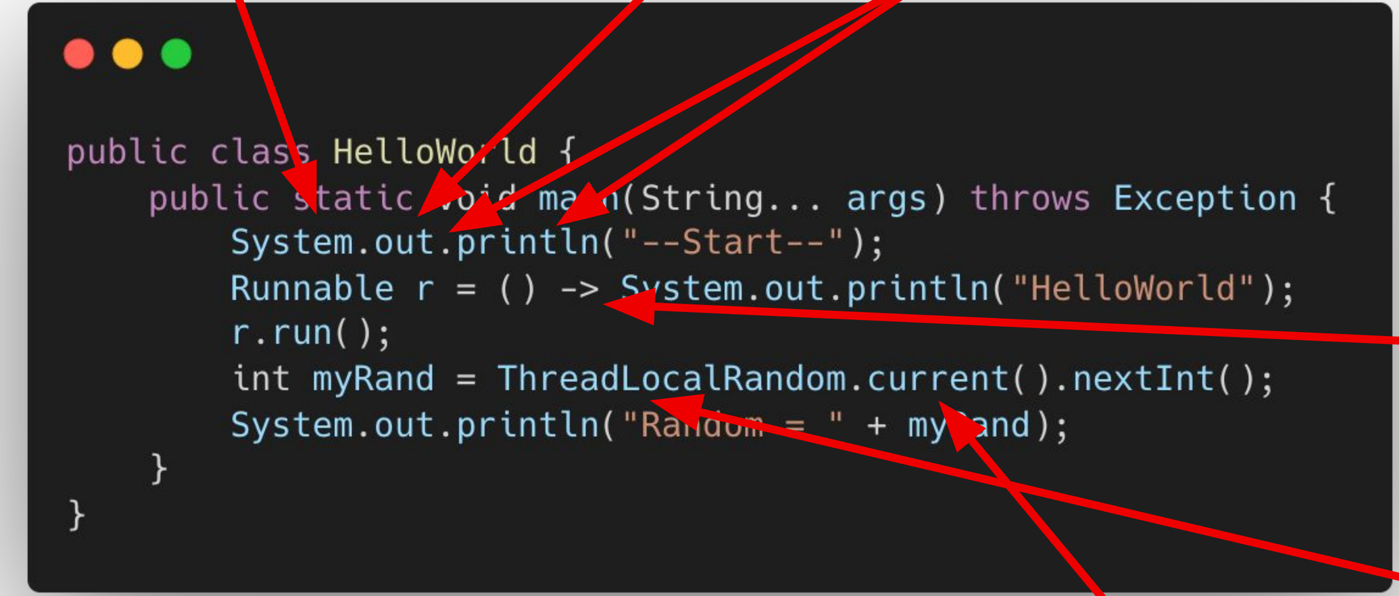
https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Dynamic Island

Dynamic classloading

Class initialization

Field and Method resolution



```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

MethodHandles
Reflection
Dynamic class generation

Dynamic classloading

Class initialization

Interpretation
Profiling
Dynamic compilation (and recompilation)

Agents
- Redefining classes
- Hooking events

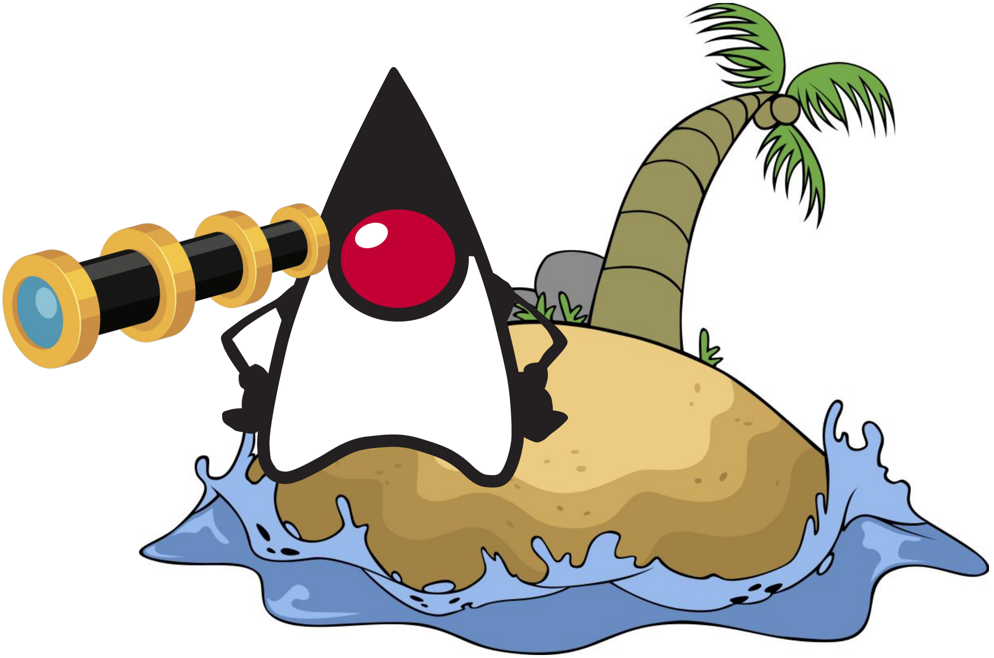
https://friendlystock.com/tropical-island-free-vector-clipart/

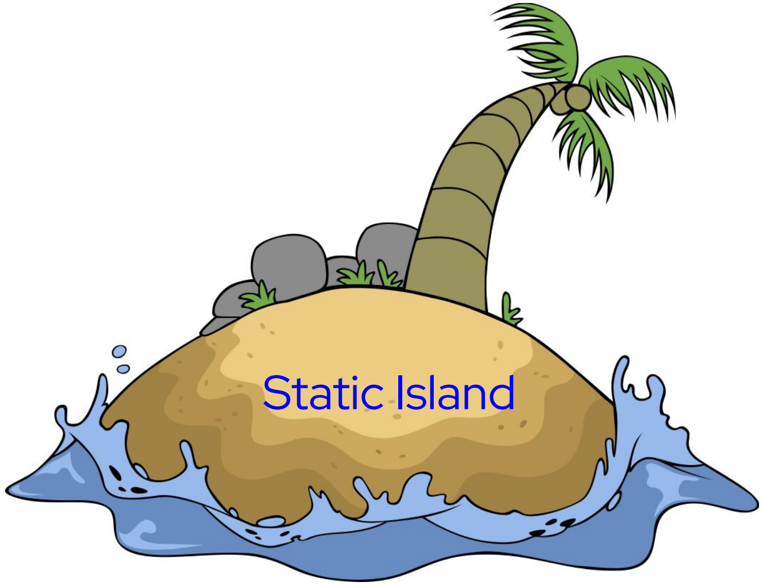
Java: Dynamic Island

Java's extremely dynamic nature is partly to blame for the "slow to start" complaints

https://friendlystock.com/tropical-island-free-vector-clipart/

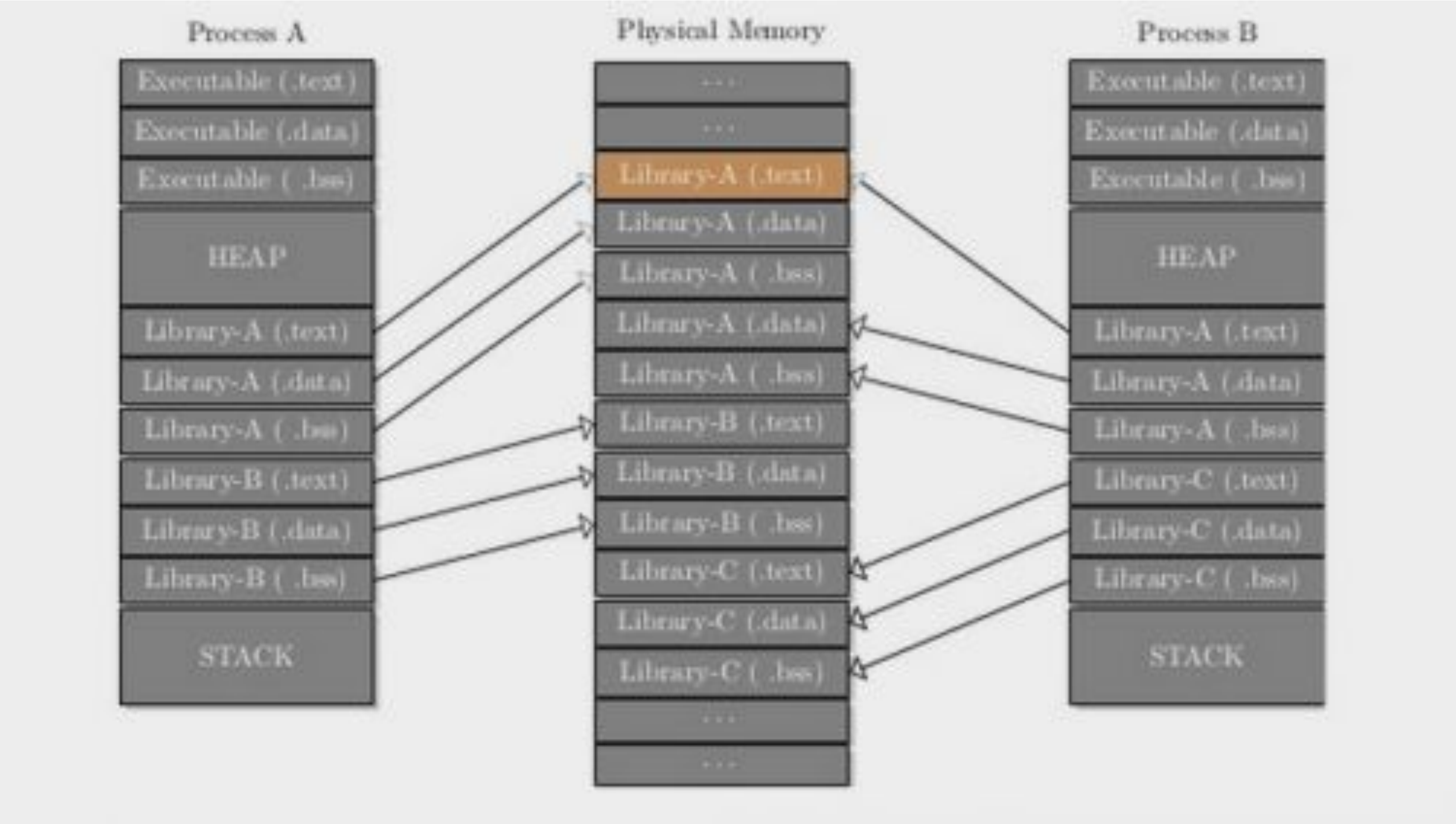
Always a little jealous of static island



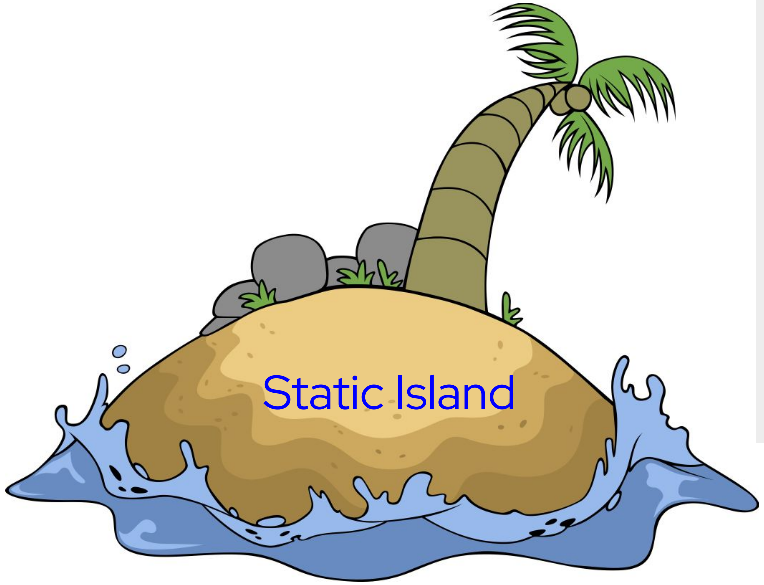


https://friendlystock.com/tropical-island-free-vector-clipart/
https://vectorportal.com/vector/spyglass-icon-vector-clip-art/
https://wiki.openjdk.org/display/duke/Gallery?preview=/http%3A%2F%2Fcr.openjdk.java.net%2F~jeff%2FDuke%2Fpng%2FHips.png

Originally because of footprint!

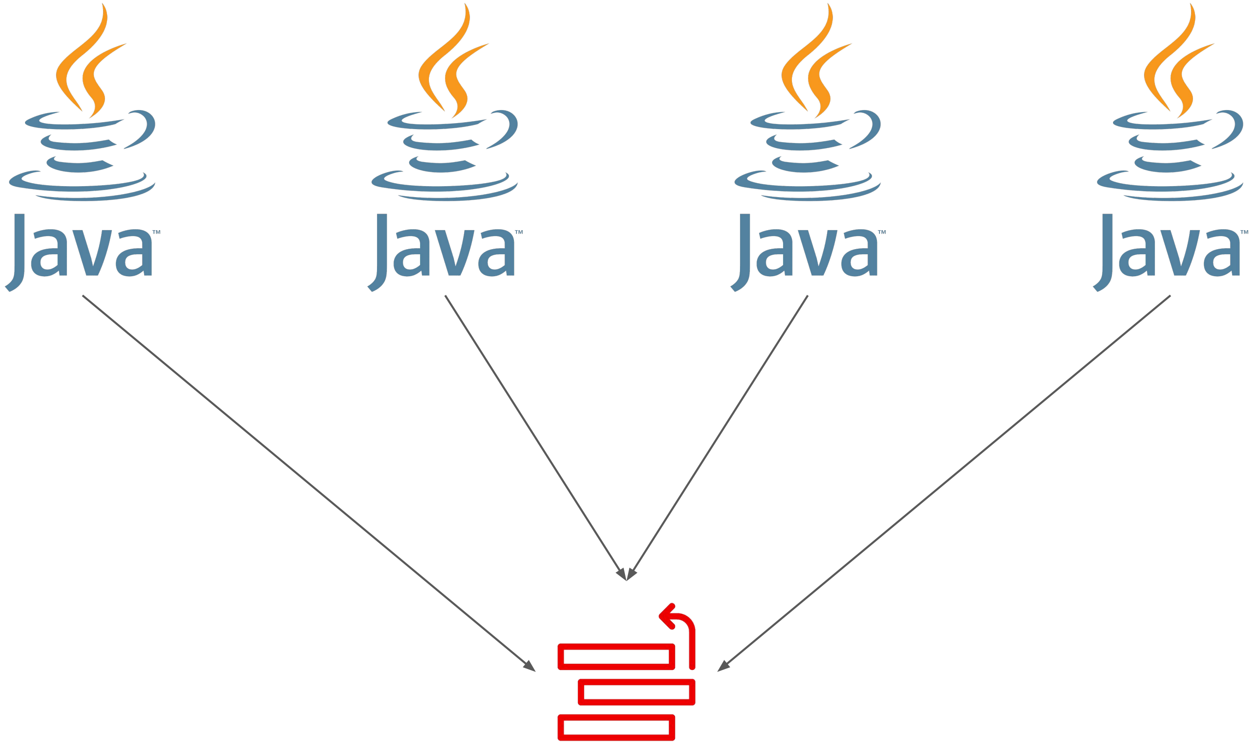


https://security.cs.pub.ro/summer-school/wiki/session/03



https://friendlystock.com/tropical-island-free-vector-clipart/

Shared Class MetaData



N * JVMs + 1 * Shared MetaData = memory footprint savings

Meta data archives enable other optimizations

Start using these features today if not already!

OpenJDK

CDS / AppCDS / DynamicCDS

Pregenerated list of classes

Dynamic set of classes at shutdown

And cached Java Objects for faster startup

OpenJ9

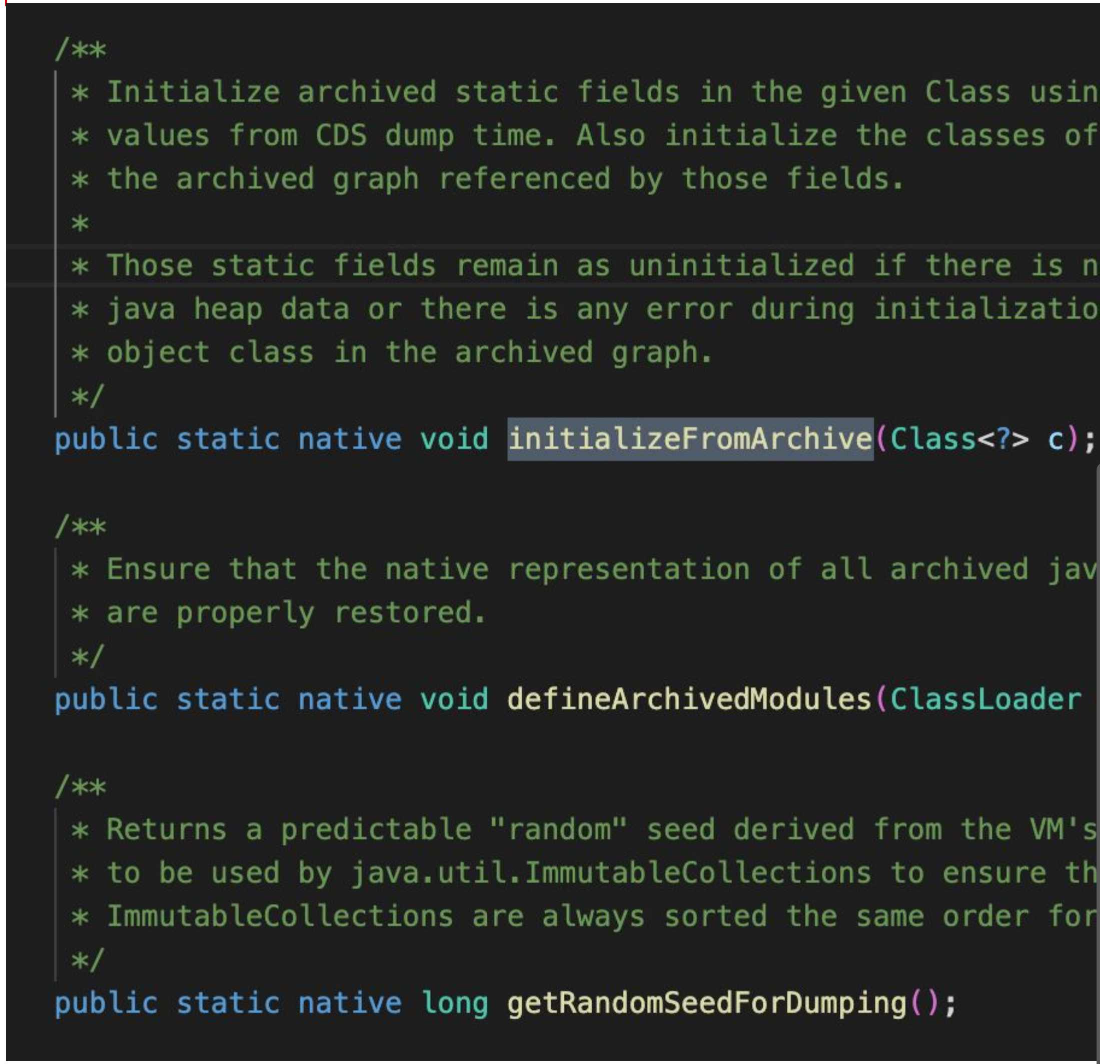
SharedClasses

Dynamic set of classes from

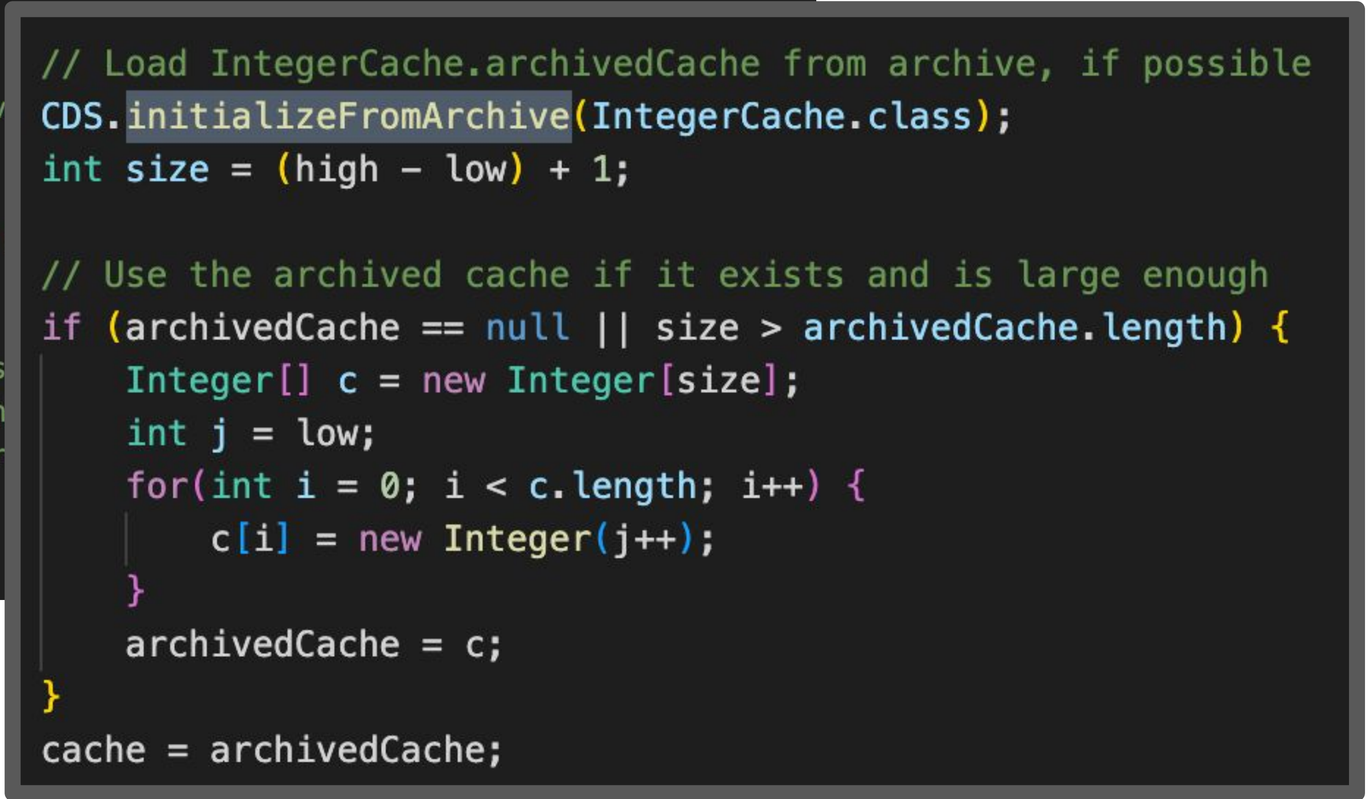
- default loaders,
- URLClassloader and
- from opted-in custom loaders

And dynamic AOT for faster startup

CDS: Archived Heaps

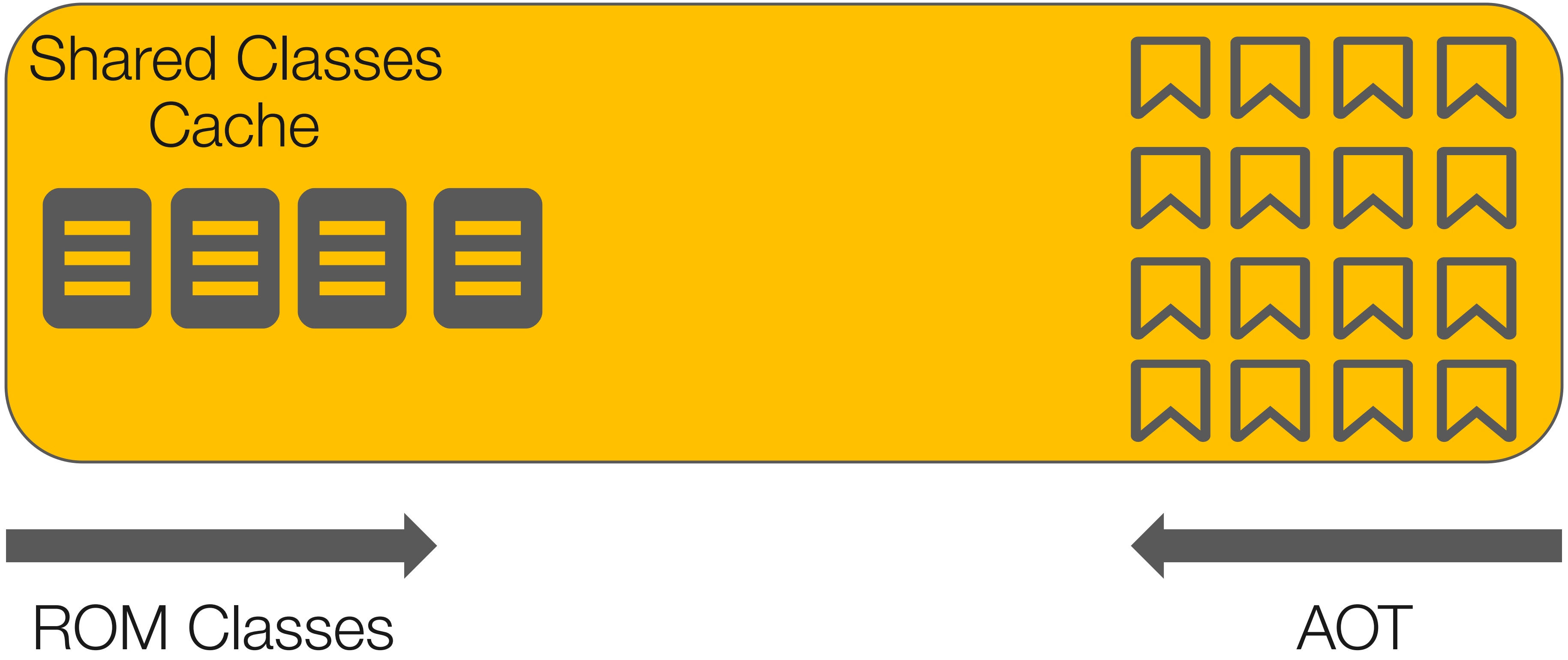


```
/**
 * Initialize archived static fields in the given Class using archived
 * values from CDS dump time. Also initialize the classes of objects in
 * the archived graph referenced by those fields.
 *
 * Those static fields remain as uninitialized if there is no mapped CDS
 * java heap data or there is any error during initialization of the
 * object class in the archived graph.
 */
public static native void initializeFromArchive(Class<?> c);

/**
 * Ensure that the native representation of all archived jav
 * are properly restored.
 */
public static native void defineArchivedModules(ClassLoader

/**
 * Returns a predictable "random" seed derived from the VM's
 * to be used by java.util.ImmutableCollections to ensure th
 * ImmutableCollections are always sorted the same order for
 */
public static native long getRandomSeedForDumping();
```



```
// Load IntegerCache.archivedCache from archive, if possible
CDS.initializeFromArchive(IntegerCache.class);
int size = (high - low) + 1;

// Use the archived cache if it exists and is large enough
if (archivedCache == null || size > archivedCache.length) {
    Integer[] c = new Integer[size];
    int j = low;
    for(int i = 0; i < c.length; i++) {
        c[i] = new Integer(j++);
    }
    archivedCache = c;
}
cache = archivedCache;
```

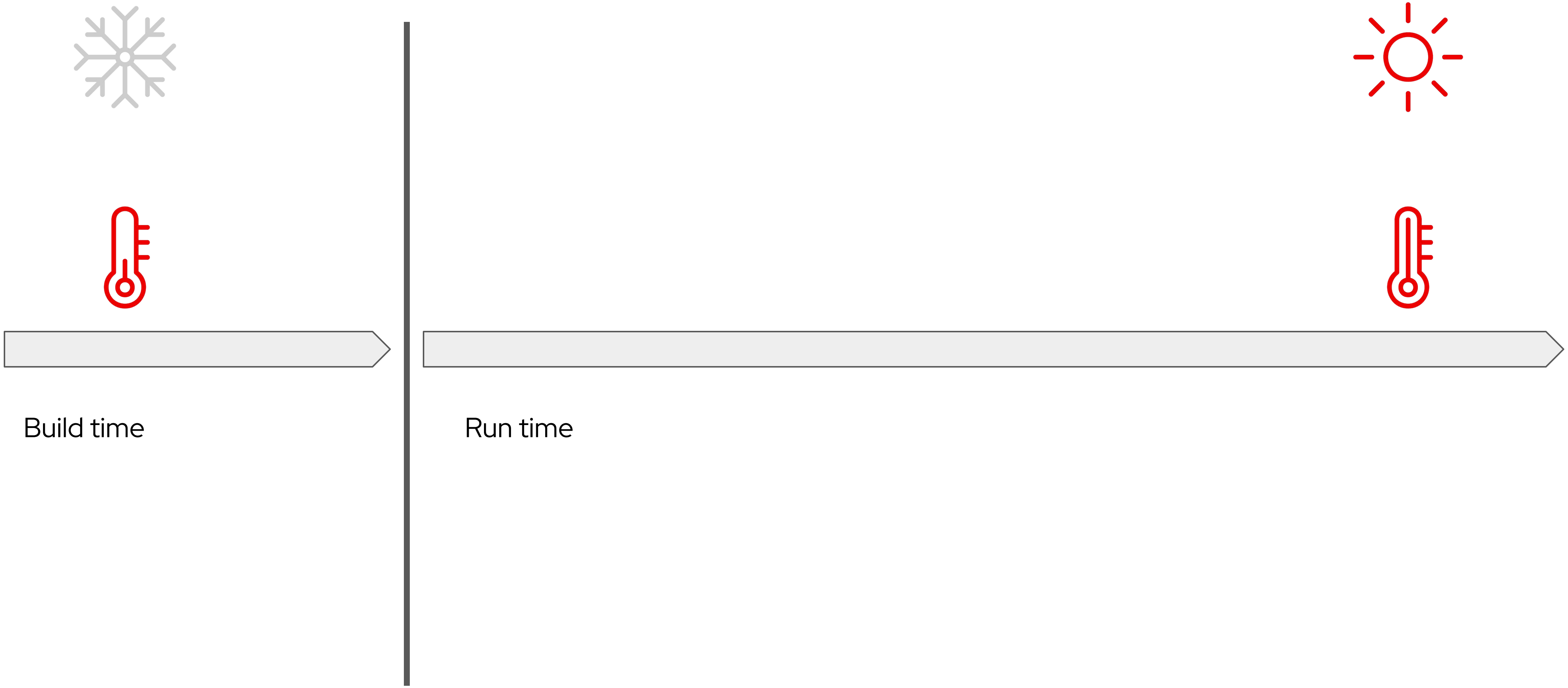
SharedClasses: dynamic AOT



```
$ java -Xshareclasses ...
```

New phases: cold vs warm runs

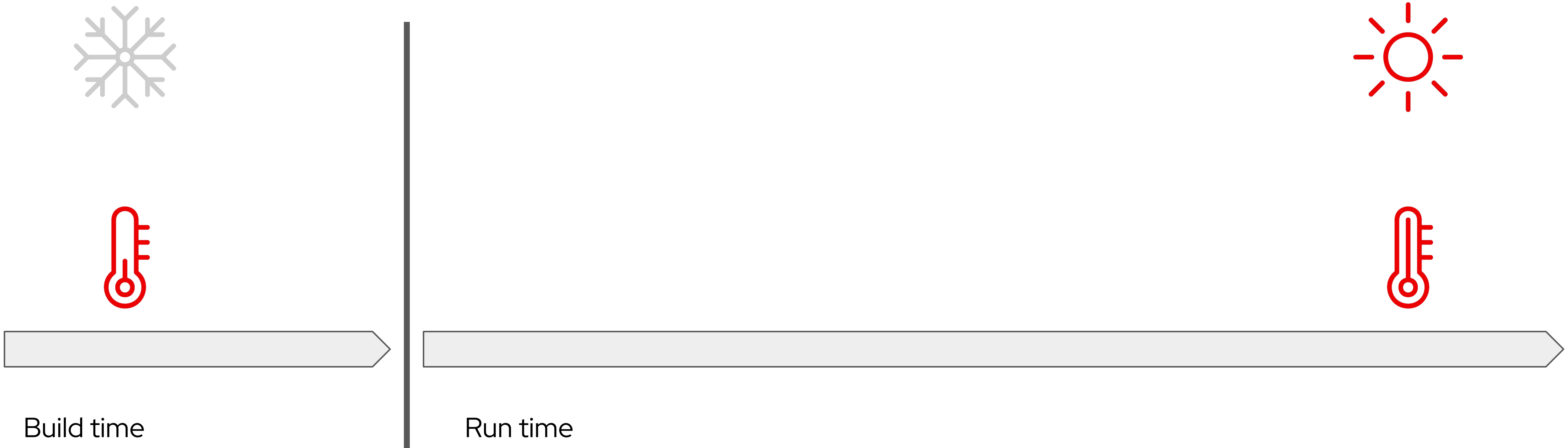
New phases: cold vs warm runs



Build time

Run time

New phases: cold vs warm runs



Build time

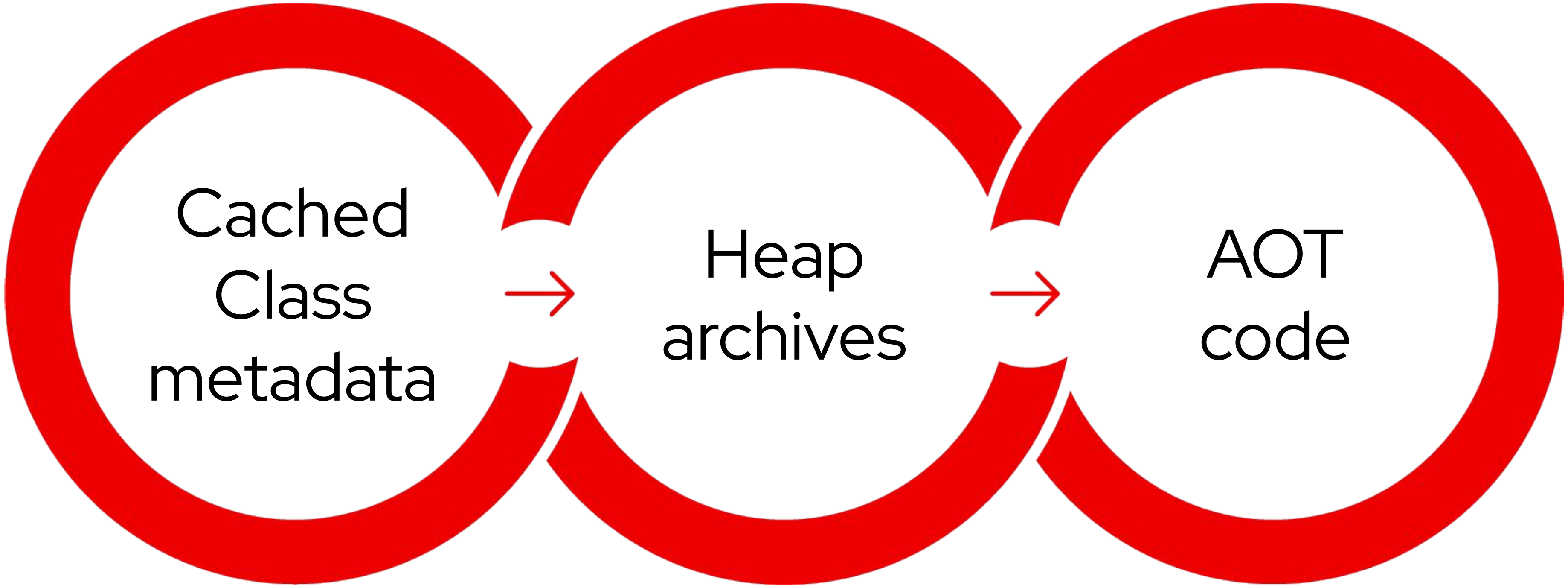
Run time

Container deployments are immutable!
Every run is a “first run”
No warm runs apart from what happened at build time

Using these tools involves more work in the build phase!

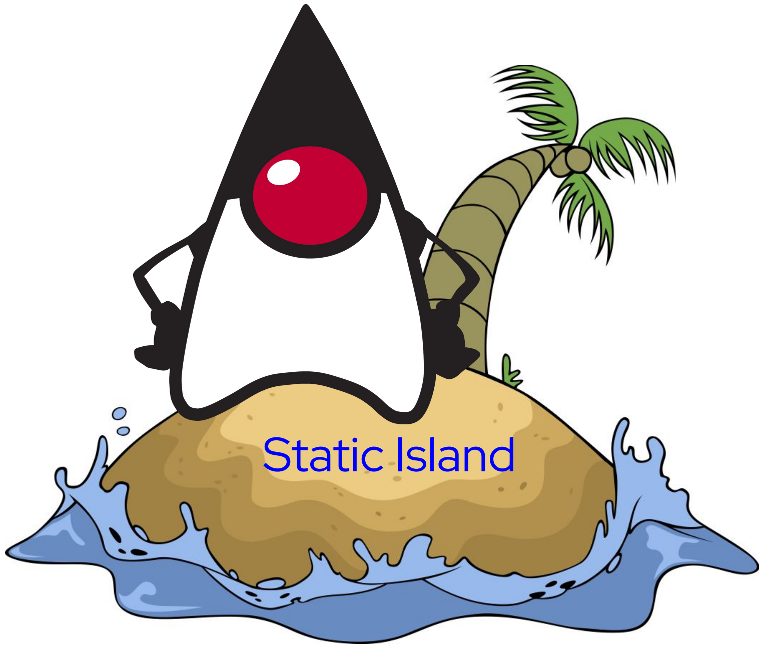
The three essentials for fast startup

Starting to appear in various forms!



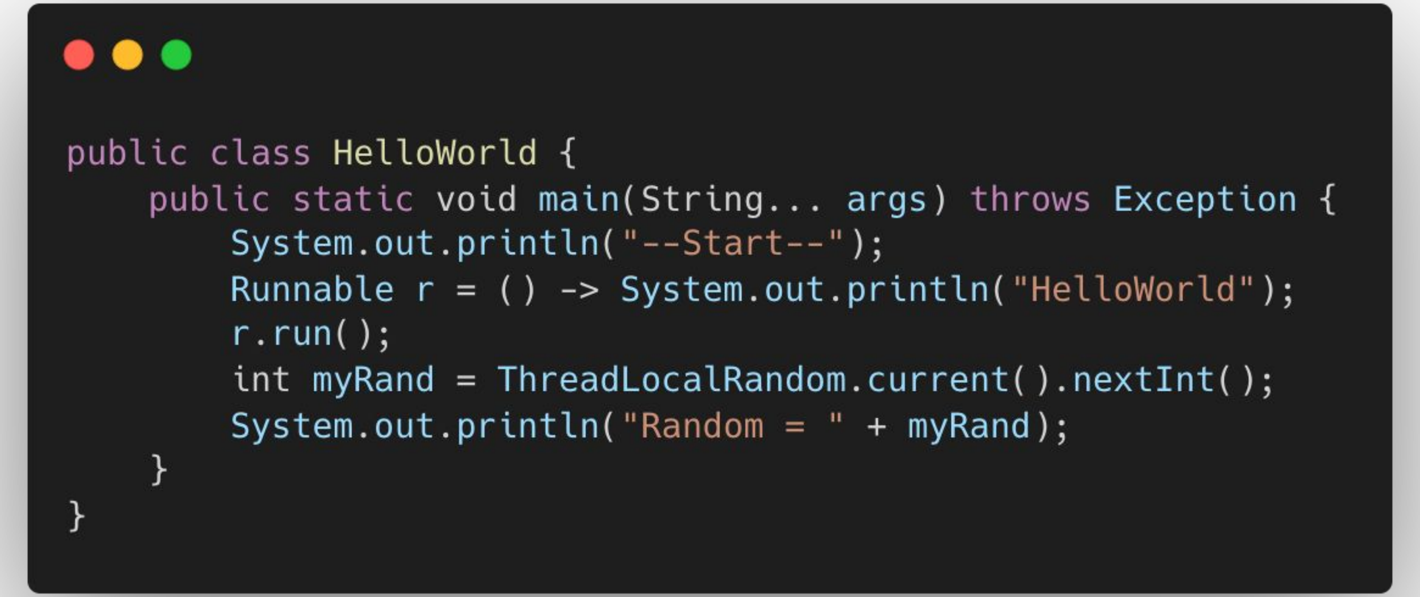
Cached Class metadata → Heap archives → AOT code

What would static Java look like?



https://friendlystock.com/tropical-island-free-vector-clipart/
https://wiki.openjdk.org/display/duke/Gallery?preview=/http%3A%2F%2Fcr.openjdk.java.net%2F~jeff%2FDuke%2Fpng%2FHips.png

Java: Static Island?



```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

https://friendlystock.com/tropical-island-free-vector-clipart/

Java: Static Island?

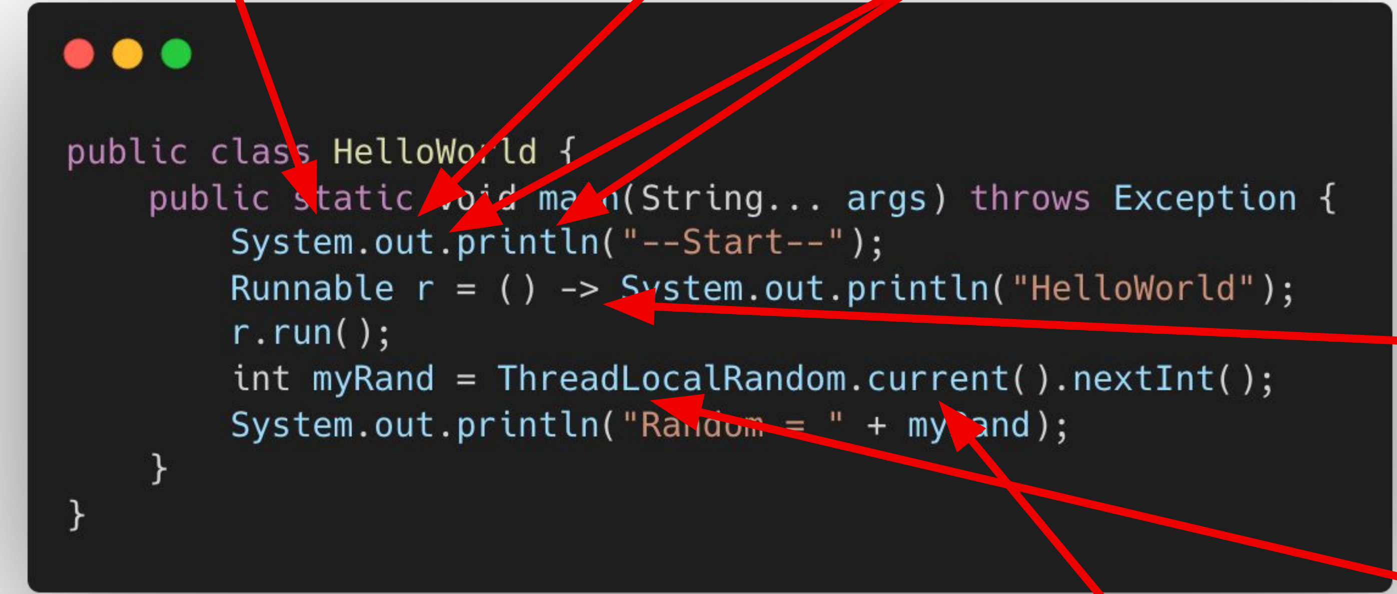
BuildTime Classloading

~~Dynamic Classloading~~

Class initialization???

Everything resolved at compile time

~~Field and Method resolution~~



```
public class HelloWorld {
    public static void main(String... args) throws Exception {
        System.out.println("--Start--");
        Runnable r = () -> System.out.println("HelloWorld");
        r.run();
        int myRand = ThreadLocalRandom.current().nextInt();
        System.out.println("Random = " + myRand);
    }
}
```

MethodHandles
Reflection
~~Dynamic Class generation~~

Needs to be pre-configured

~~Dynamic Classloading~~

Class initialization???

Service providers?

Class generators?

Module Layers?

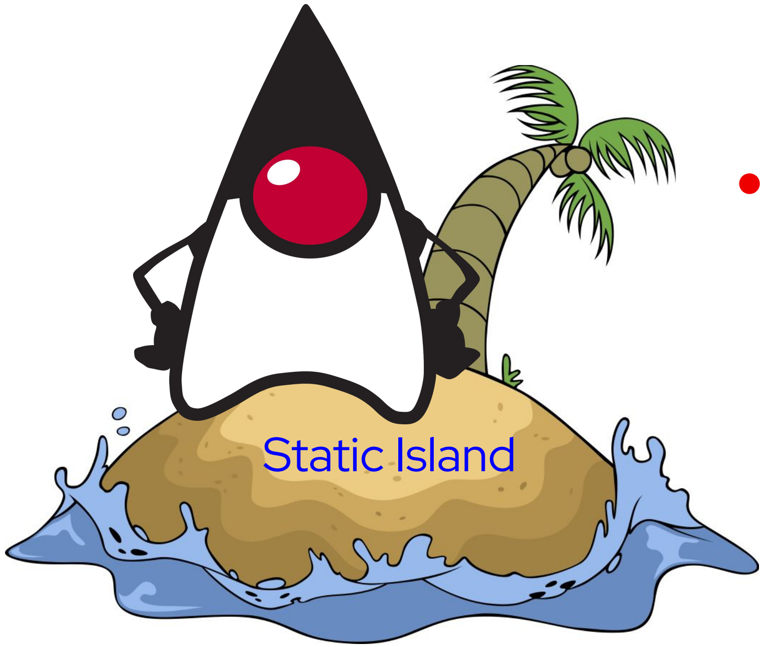
~~Interpretation~~
~~Profiling~~
~~Dynamic compilation~~ (and recompilation)

Ahead of time compiled

~~Agents~~
~~Redefining classes~~
~~Hooking events~~

https://friendlystock.com/tropical-island-free-vector-clipart/

What would static Java look like?

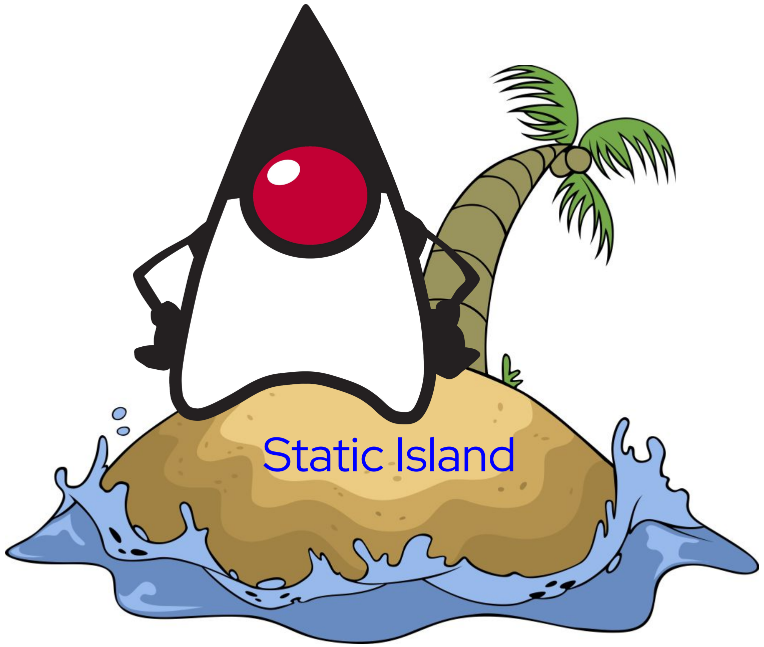


- All classes – application & class library – available at build time
  - No runtime class generation!
  - Closed world
  - Classloaders?

- Compiled to native code at build time
  - No decompilation / recompilation
  - Class initialization checks remain in the compiled code
  - AOT “guesses” about what will happen... and where it will run
  - Profiling?

https://friendlystock.com/tropical-island-free-vector-clipart/
https://wiki.openjdk.org/display/duke/Gallery?preview=/http%3A%2F%2Fcr.openjdk.java.net%2F~jeff%2FDuke%2Fpng%2FHips.png

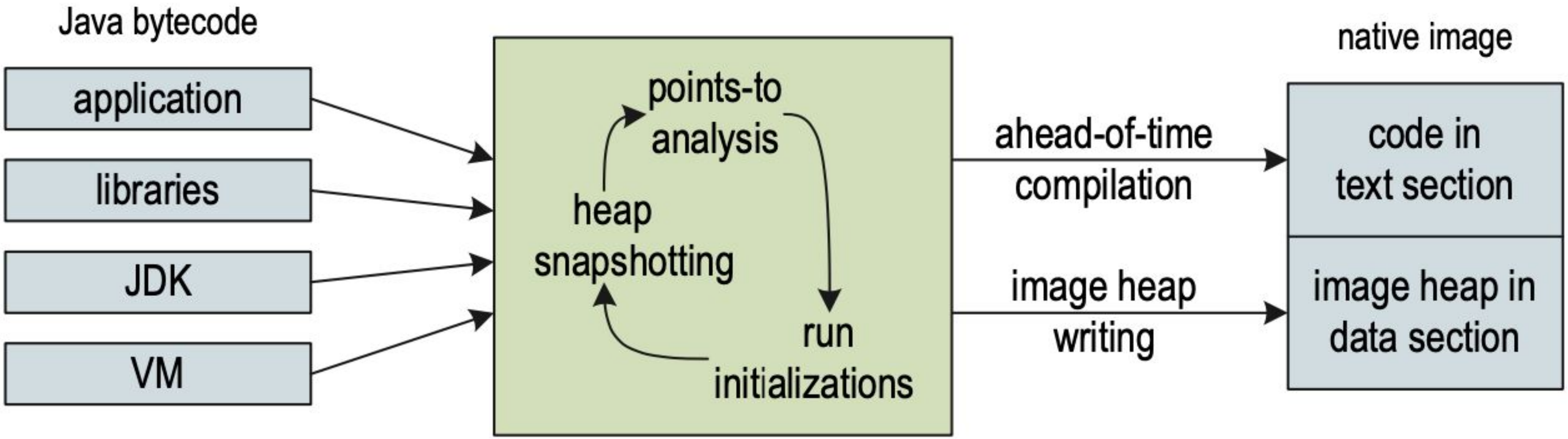
Static Java challenges



- What about classloaders?
  - Classes are available at build time… only class/module path?
  - How does this work with application specific loaders?
  - No (runtime) generated code!
- Native code is 3-5x larger than bytecode
  - Need some way to trim dead code
  - Without removing indirectly accessed code (reflection / methodhandles)
- And it would be nice to initialize some of those classes at buildtime

https://friendlystock.com/tropical-island-free-vector-clipart/
https://wiki.openjdk.org/display/duke/Gallery?preview=/http%3A%2F%2Fcr.openjdk.java.net%2F~jeff%2FDuke%2Fpng%2FHips.png

GraalVM™

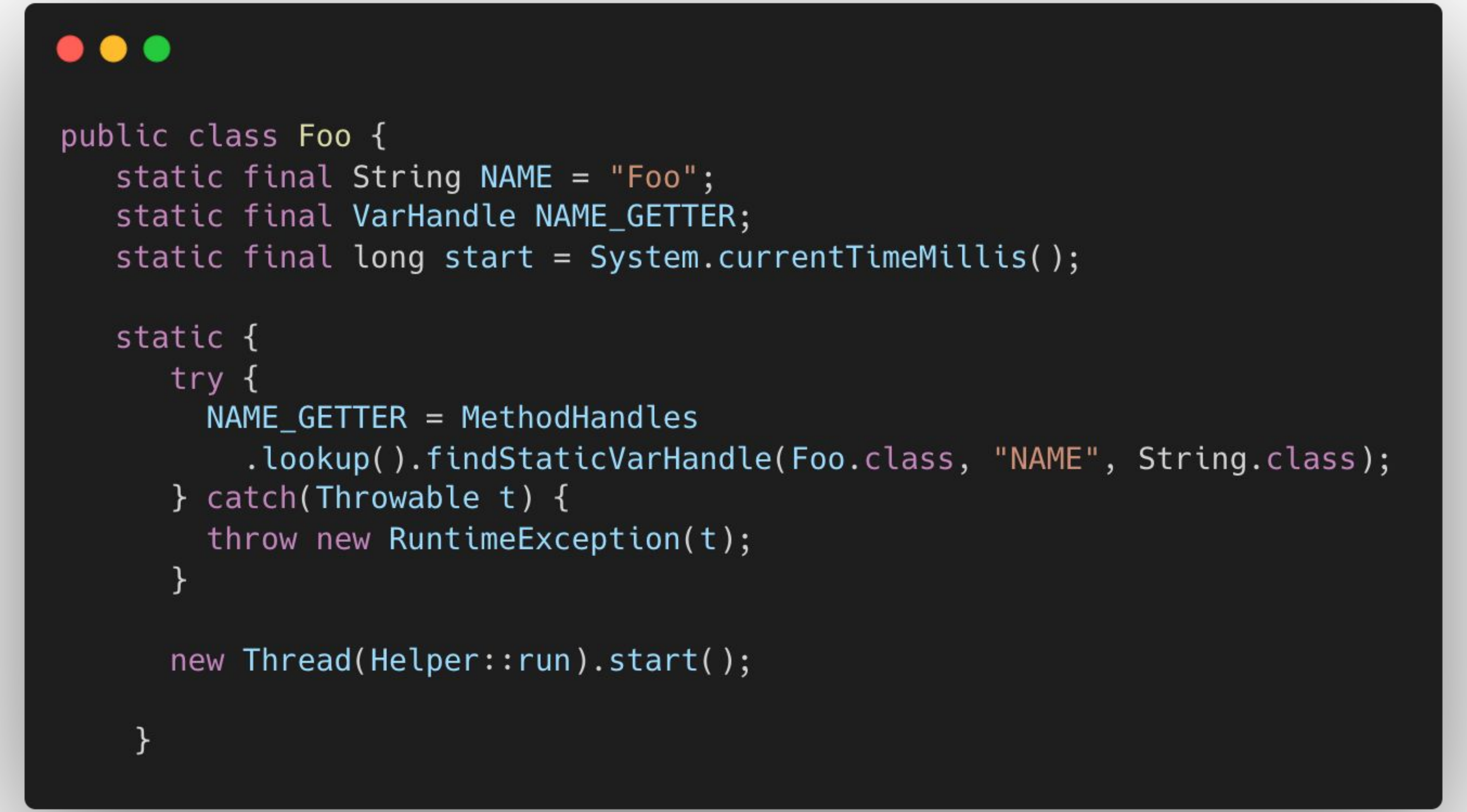


Virtuous cycle:

- Points-to analysis allows dead code elimination (DCE)
- Heap snapshotting allows initializing code at buildtime
- Running class initialization at buildtime allows more DCE (ie: <clinit> methods)
- ... and repeat

Source: Initialize Once, Start Fast: Application Initialization at Build Time (Wimmer et al. OOPSLA 2019)

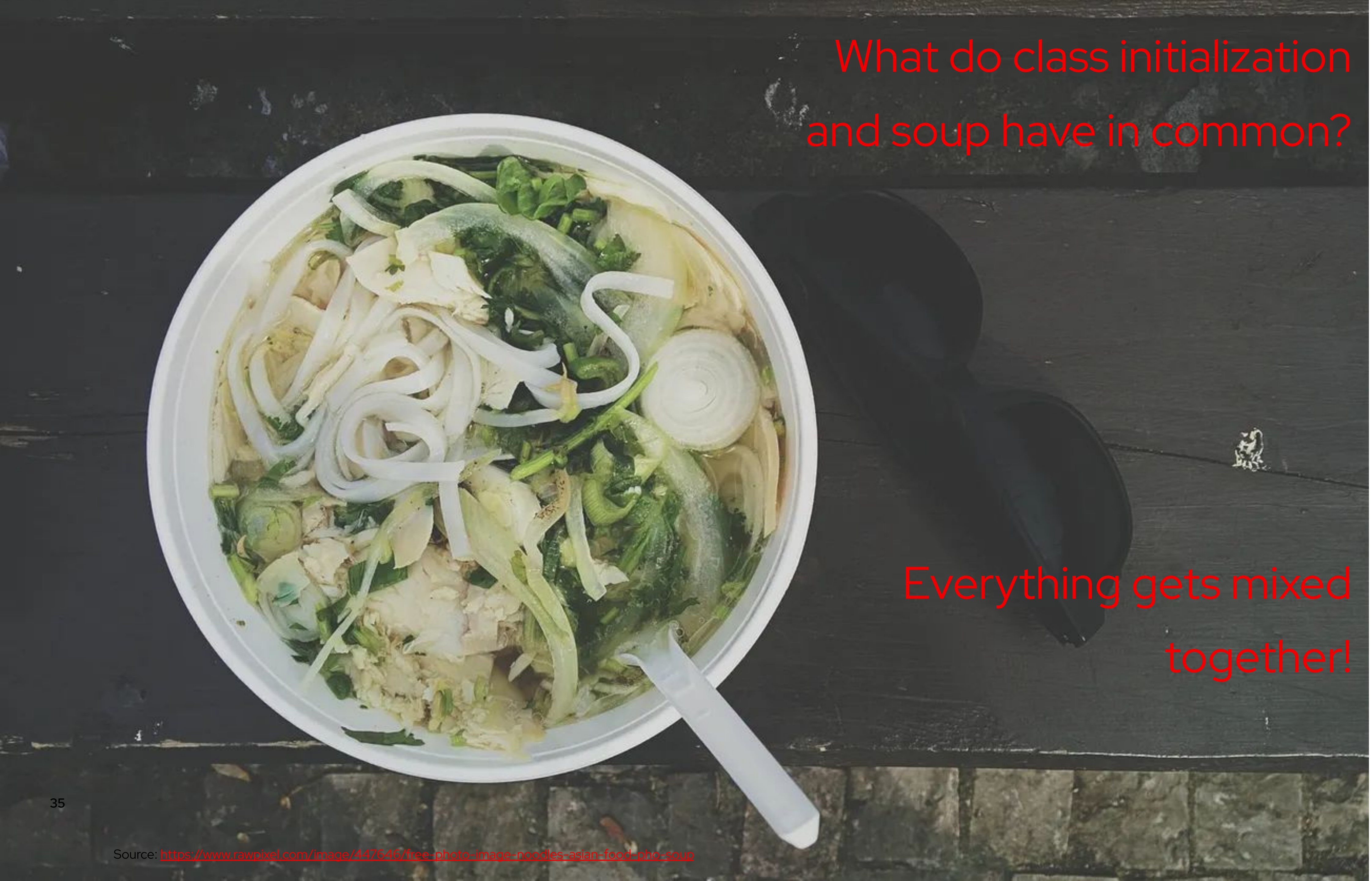
Static Java: the question of class initialization



```
public class Foo {
   static final String NAME = "Foo";
   static final VarHandle NAME_GETTER;
   static final long start = System.currentTimeMillis();

   static {
      try {
        NAME_GETTER = MethodHandles
          .lookup().findStaticVarHandle(Foo.class, "NAME", String.class);
      } catch(Throwable t) {
        throw new RuntimeException(t);
      }

      new Thread(Helper::run).start();

    }
```

- When to initialize this class?
  - Buildtime?
  - Runtime?
  - Both (aka re-initialize)?
- Developer needs to decide for each class when it should be initialized
- Default was buildtime, then runtime, soon to be reinitialize

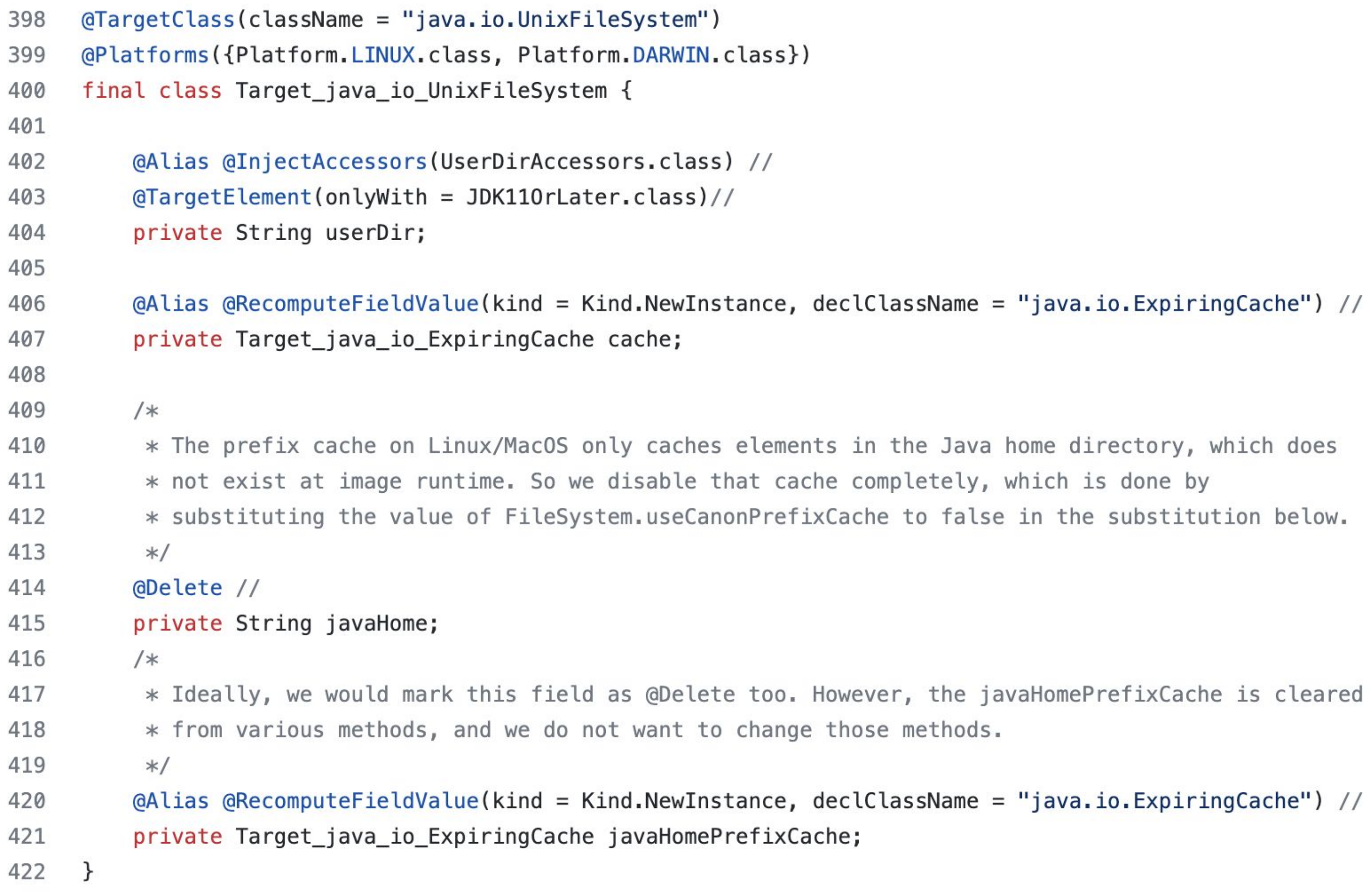
What do class initialization and soup have in common?



Everything gets mixed together!

Source: https://www.rawpixel.com/image/447646/free-photo-image-noodles-asian-food-pho-soup

Static Java: Substitutions

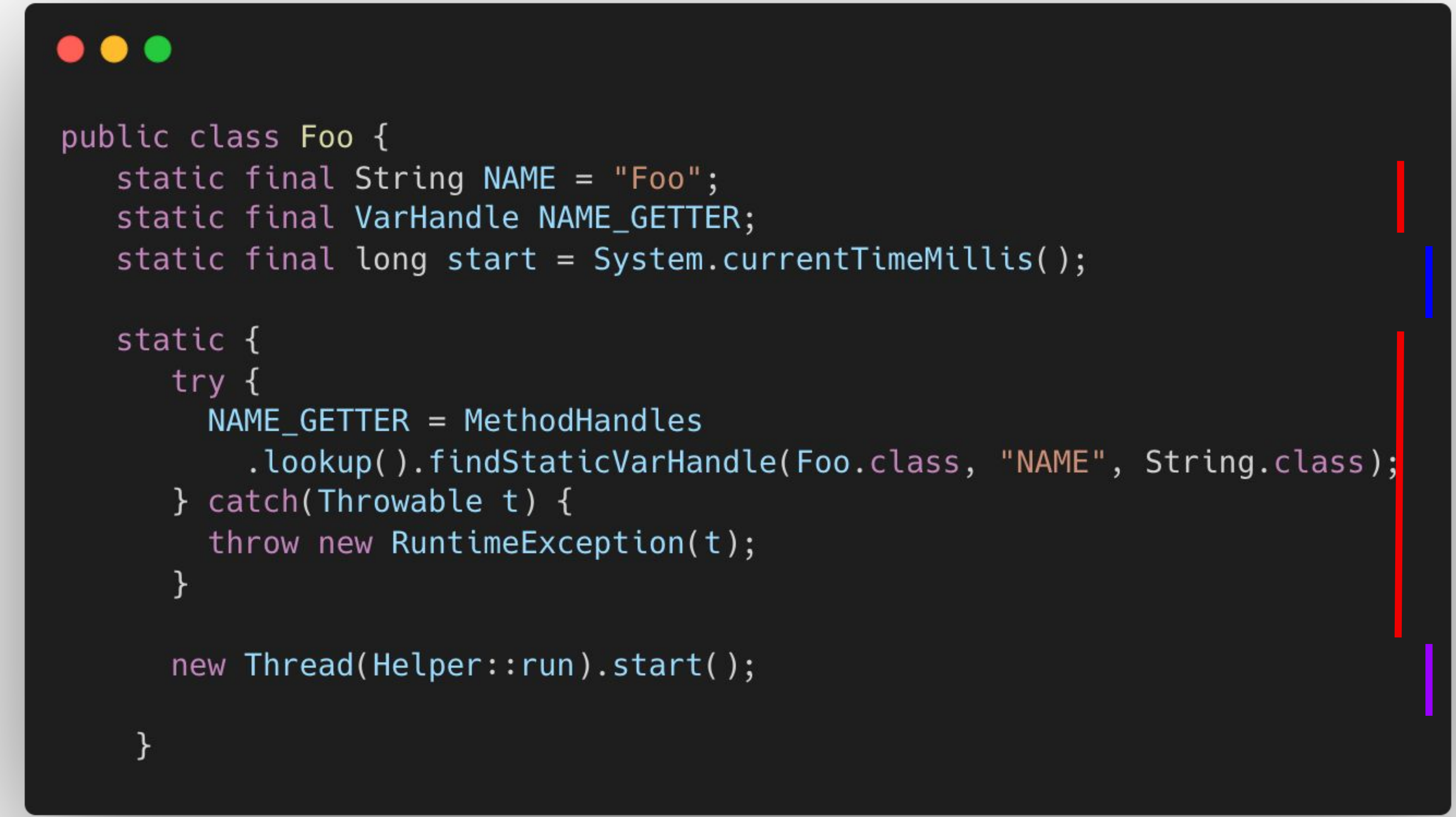


```java
@TargetClass(className = "java.io.UnixFileSystem")
@Platforms({Platform.LINUX.class, Platform.DARWIN.class})
final class Target_java_io_UnixFileSystem {

    @Alias @InjectAccessors(UserDirAccessors.class) //
    @TargetElement(onlyWith = JDK11OrLater.class)//
    private String userDir;

    @Alias @RecomputeFieldValue(kind = Kind.NewInstance, declClassName = "java.io.ExpiringCache") //
    private Target_java_io_ExpiringCache cache;

    /*
     * The prefix cache on Linux/MacOS only caches elements in the Java home directory, which does
     * not exist at image runtime. So we disable that cache completely, which is done by
     * substituting the value of FileSystem.useCanonPrefixCache to false in the substitution below.
     */
    @Delete //
    private String javaHome;
    /*
     * Ideally, we would mark this field as @Delete too. However, the javaHomePrefixCache is cleared
     * from various methods, and we do not want to change those methods.
     */
    @Alias @RecomputeFieldValue(kind = Kind.NewInstance, declClassName = "java.io.ExpiringCache") //
    private Target_java_io_ExpiringCache javaHomePrefixCache;
}
```

- Essential mechanism to "fix" code to support buildtime initialization
- Easy to change the meaning of the program by breaking invariants
    - Dynamic vs static disparity
- Easy to get out of sync as maintained separately from the code it modifies

Source:
https://github.com/oracle/graal/blob/a5ca5bda7301447bb2c755134f8403e51621dfa4/substratevm/src/com.oracle.svm.core/src/com/oracle/svm/core/jdk/FileSystemProviderSupport.java

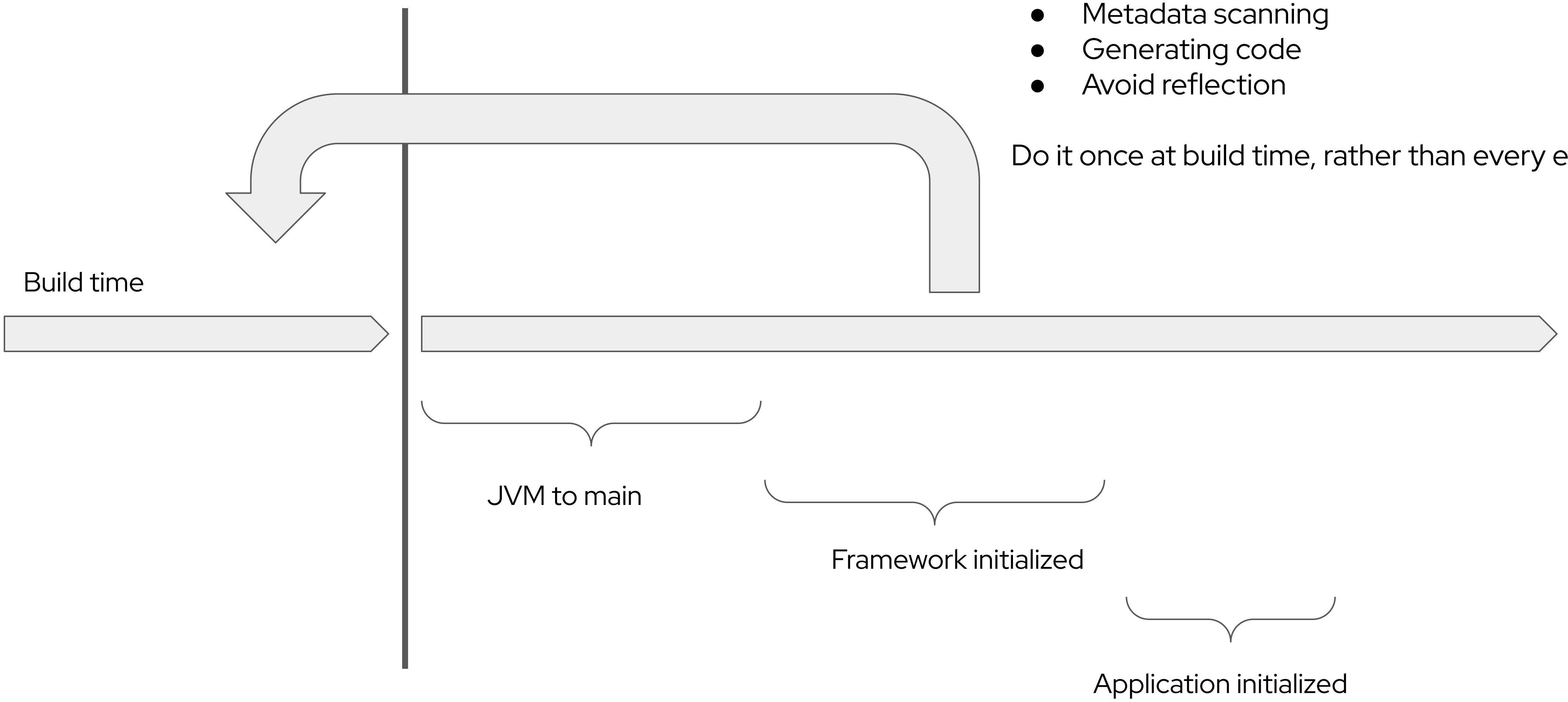
qbicc: Class initialization example



```
public class Foo {
   static final String NAME = "Foo";
   static final VarHandle NAME_GETTER;
   static final long start = System.currentTimeMillis();

   static {
      try {
        NAME_GETTER = MethodHandles
          .lookup().findStaticVarHandle(Foo.class, "NAME", String.class);
      } catch(Throwable t) {
        throw new RuntimeException(t);
      }

      new Thread(Helper::run).start();

    }
```

- Build time initialized
- Per-field <rtinit> re-initialize method
- $_patch class to move Thread::start

Frameworks: Static Java’s best friend



Favour build time work over runtime work

- Metadata scanning
- Generating code
- Avoid reflection

Do it once at build time, rather than every execution

QUARKUS

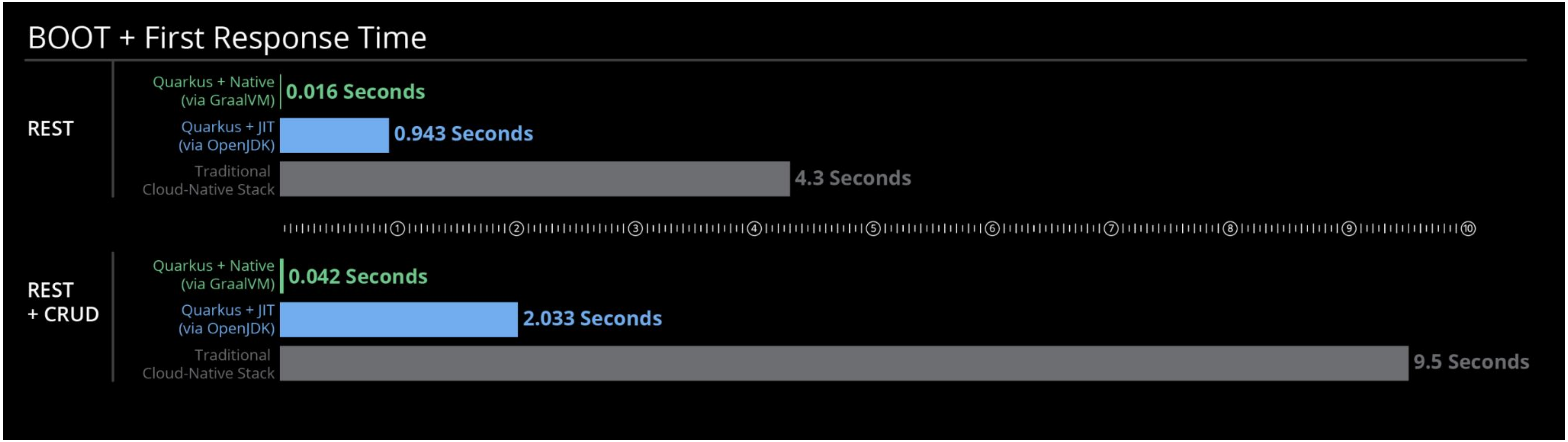
Faster JVM mode startup

+

Enabling native image startup

MICRONAUT®

Static Java: Framework results



Benefits:

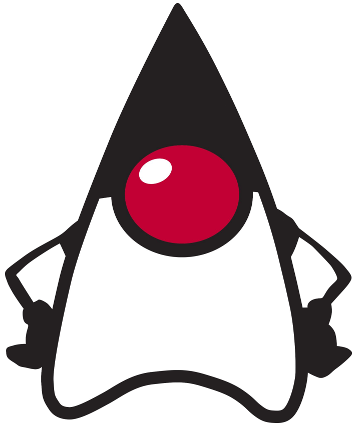
- Very fast startup
- Small on disk footprint / variable runtime footprint
- Fast time to peak perf / lower peak?

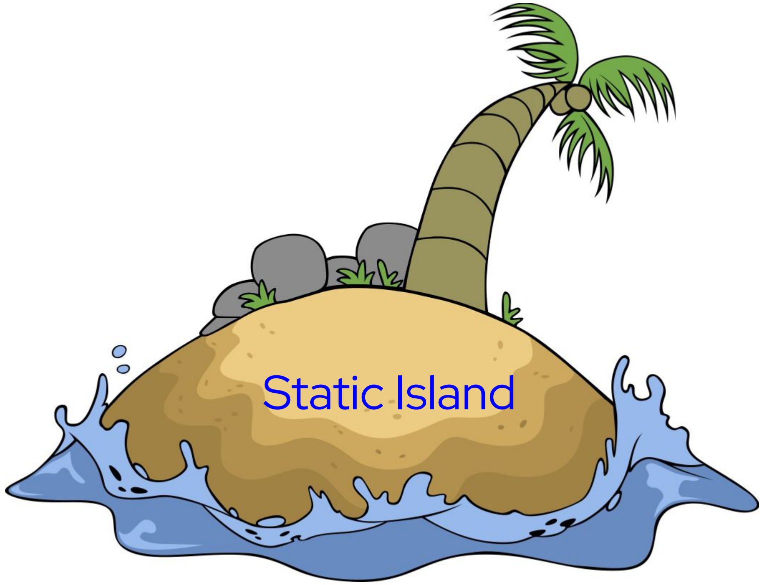
Costs:

- Closed world
- Changes (Substitutions) required
- Dynamic vs. static disparity

Source https://quarkus.io/

Is there a middle ground?





https://friendlystock.com/tropical-island-free-vector-clipart/
https://wiki.openjdk.org/display/duke/Gallery?preview=/http%3A%2F%2Fcr.openjdk.java.net%2F~jeff%2FDuke%2Fpng%2FHips.png

Checkpoint and Restore

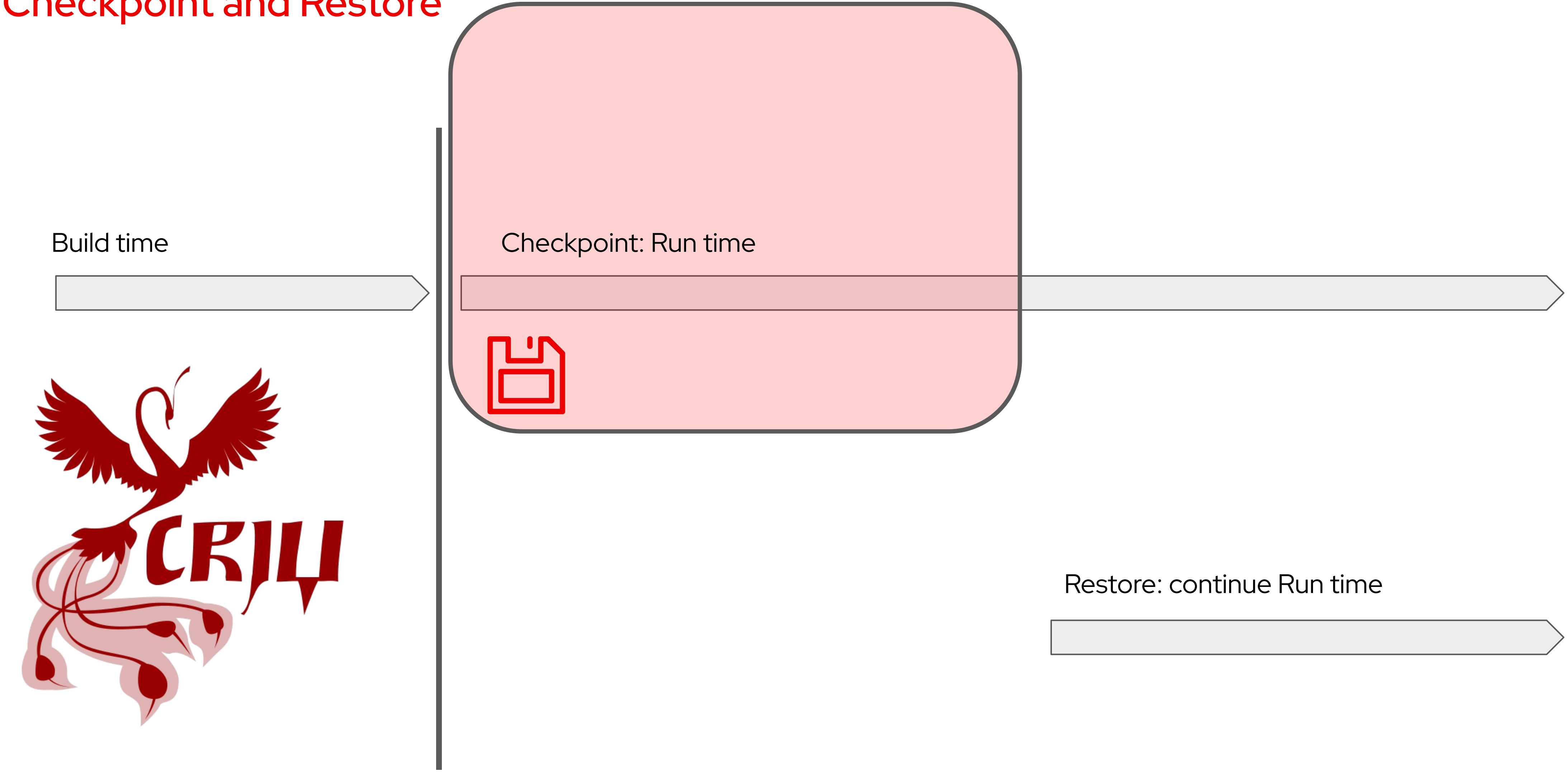
- CRIU operates at the operating system process level
  - Saves the running process to disk
  - Allows it to be restored later… many times!

- Copies ***everything*** rather than trying to provide just the 3 essentials

- Run the application to “good” point. Continue from there
  - “Good” points are application and use case specific

https://www.criu.org/Main_Page

Checkpoint and Restore



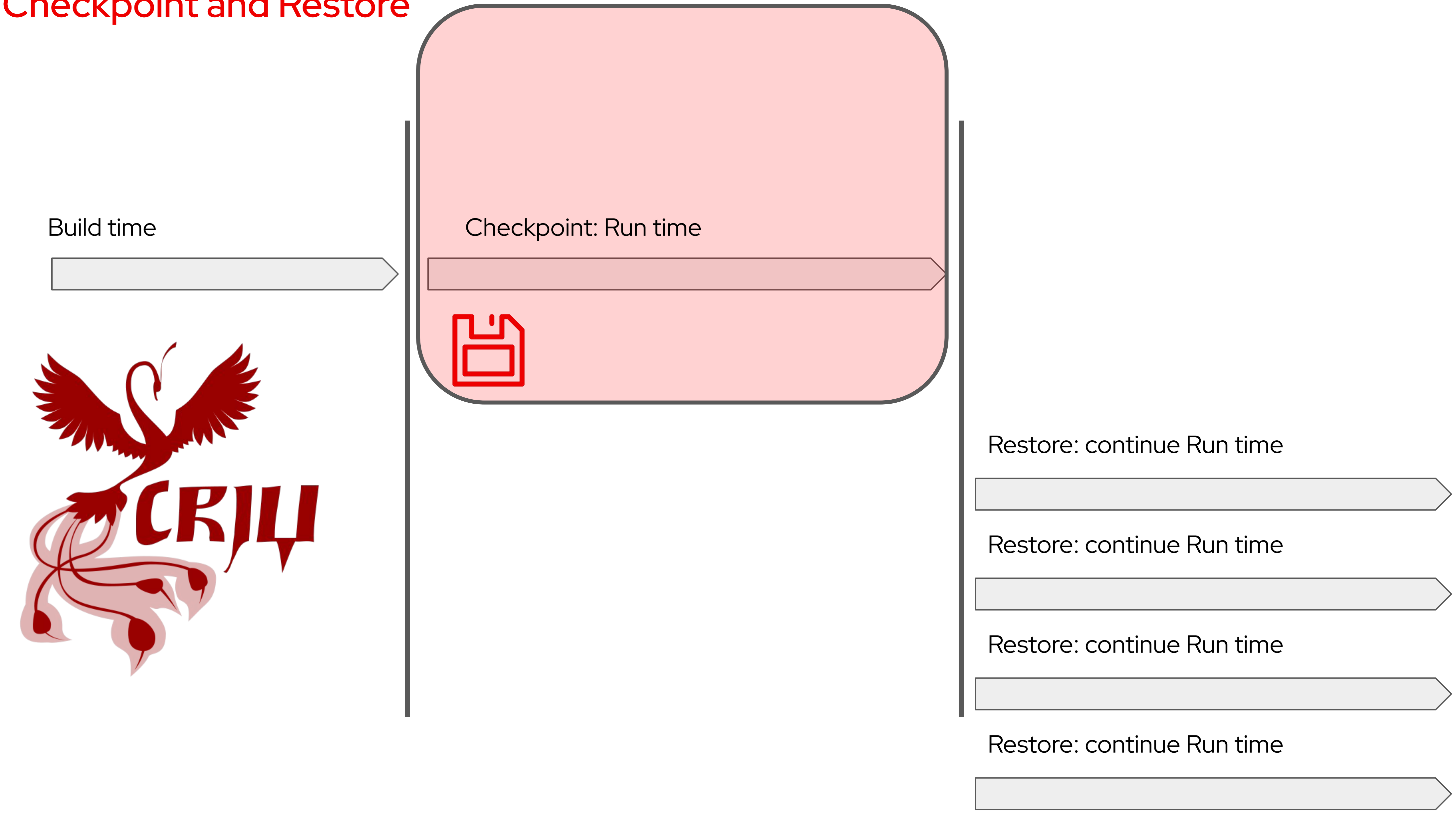
Build time

Checkpoint: Run time

Restore: continue Run time

https://www.criu.org/Main_Page

Checkpoint and Restore



Build time

Checkpoint: Run time

Restore: continue Run time

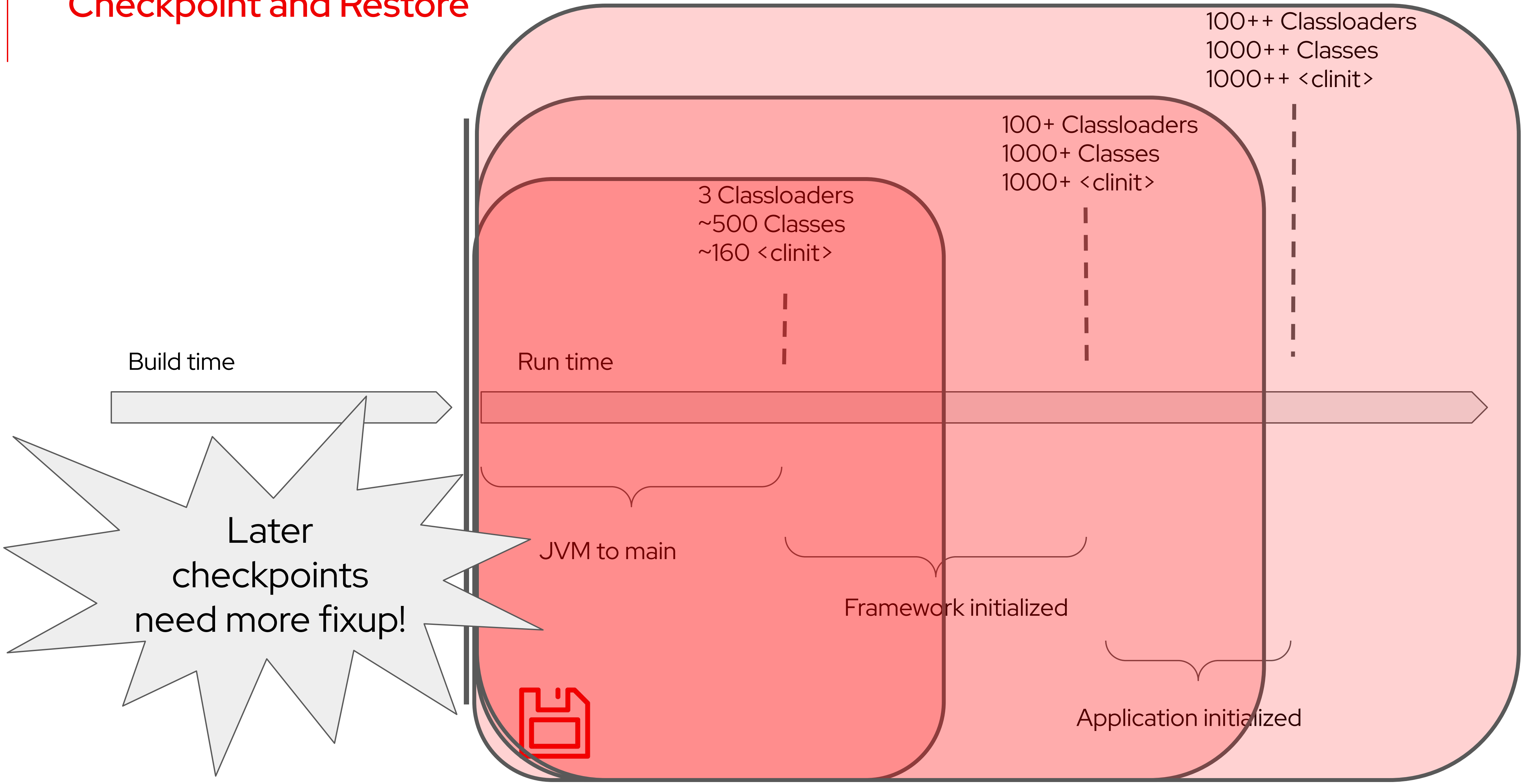
Restore: continue Run time

Restore: continue Run time

Restore: continue Run time

https://www.criu.org/Main_Page

Checkpoint and Restore



Later checkpoints need more fixup!

Checkpoint and Restore

OpenJDK

CRaC

Coordinated Restore at Checkpoint

https://openjdk.org/projects/crac/
https://github.com/CRaC

OpenJ9

CRIU Support
(aka Semeru InstantOn)

https://github.com/orgs/eclipse-openj9/projects/1

Checkpoint and Restore concerns

- Capturing the whole process also captures things you wish it didn't
  - Random / SecureRandom

  - Time deltas now include time between checkpoint and restore
    - System.nanoTime needs more care
    - System.currentTimeInMillis is just a mess

  - Environment variables may not be available till restore
    - Common deployment tool for e.g. ports, host names, etc
    - Java expects them to be immutable once fetched

  - Number of CPUs, particular CPU instructions, etc
    - Portability of the running JVM, jitted code, and libraries

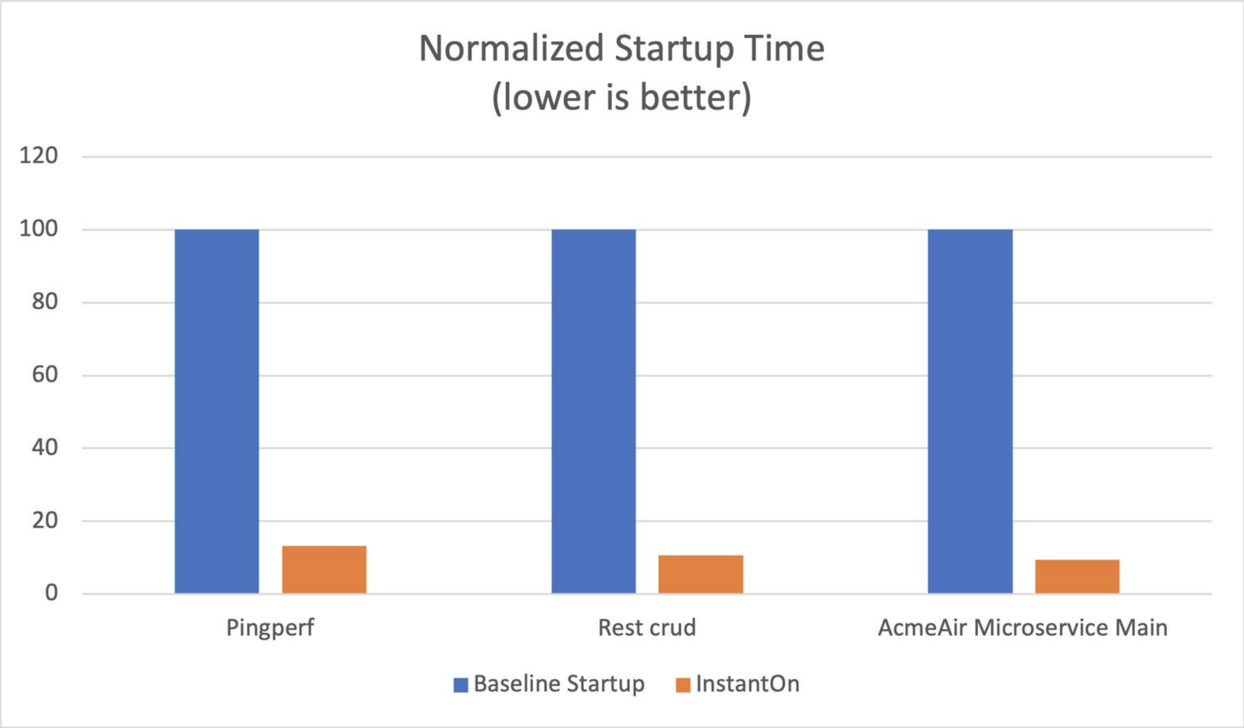
Checkpoint and Restore also needs fixups

- Project CRaC and OpenJ9 CRIU support provide Lifecycle APIs
    - Applications still need changes!
    - Callbacks to let them fix their state at checkpoint/restore
    - Core Class library addressed by the projects
- Fixups required depend on checkpoint point and use case
    - No general way to say "this is checkpoint ready"
- Not just correctness, may also provide better performance
    - JDK lazy init can be converted to pre-checkpoint init for CRIU

jdk.crac.Resource interface

CRIUSupport
::register{Pre,Post}SnapshotHook(Runnable)

Checkpoint and Restore results



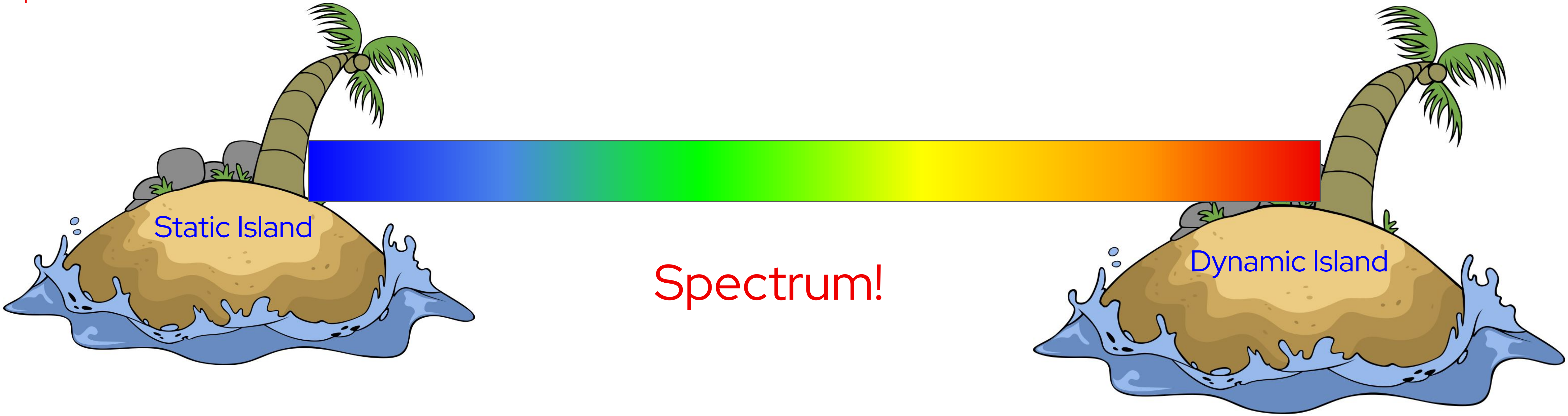
Benefits:

- Fast startup for applications that can’t opt into static Java
- Open world. Supports existing monitoring tools
- Same peak performance as dynamic JVM

Costs:

- Big on disk footprint / same runtime footprint
- Changes (Lifecycle API) required
- Pre vs Post checkpoint disparity

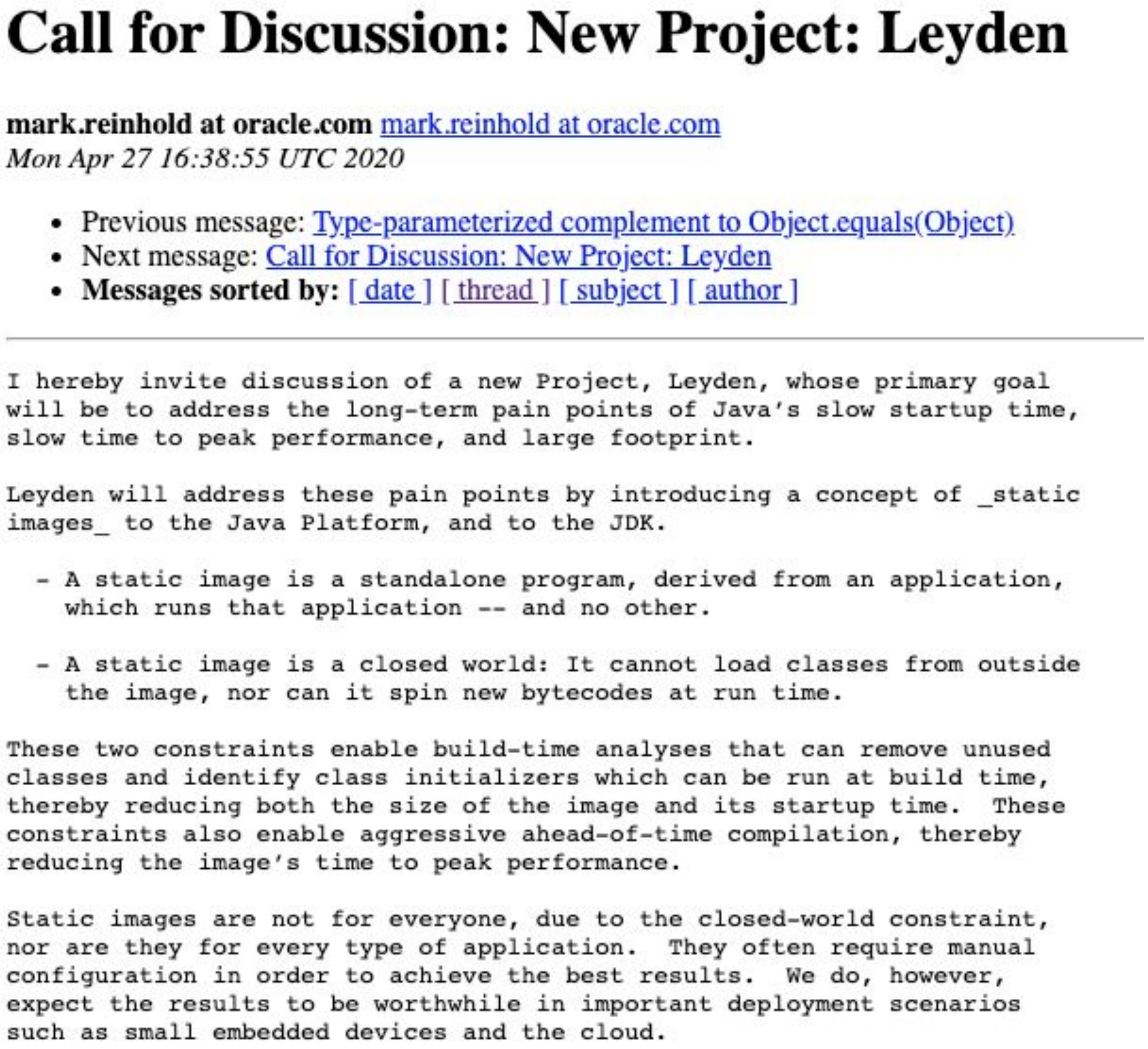
Source https://openliberty.io/blog/2022/09/29/instant-on-beta.html



Static Island

Spectrum!

Dynamic Island



Call for Discussion: New Project: Leyden

mark.reinhold at oracle.com mark.reinhold at oracle.com
Mon Apr 27 16:38:55 UTC 2020

- Previous message: Type-parameterized complement to Object.equals(Object)
- Next message: Call for Discussion: New Project: Leyden
- **Messages sorted by:** [date] [thread] [subject] [author]

I hereby invite discussion of a new Project, Leyden, whose primary goal will be to address the long-term pain points of Java's slow startup time, slow time to peak performance, and large footprint.

Leyden will address these pain points by introducing a concept of _static images_ to the Java Platform, and to the JDK.

- A static image is a standalone program, derived from an application, which runs that application -- and no other.

- A static image is a closed world: It cannot load classes from outside the image, nor can it spin new bytecodes at run time.

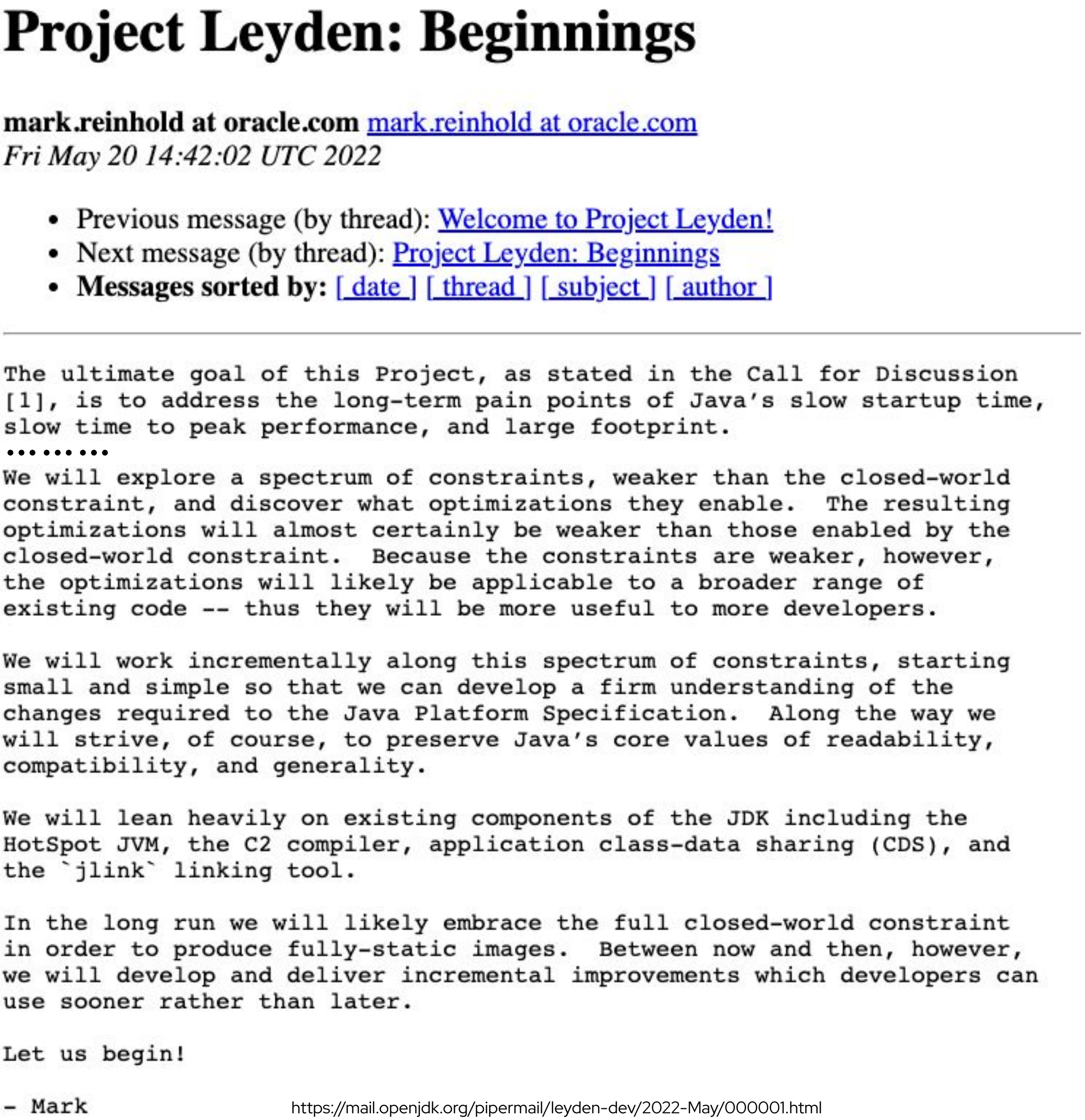
These two constraints enable build-time analyses that can remove unused classes and identify class initializers which can be run at build time, thereby reducing both the size of the image and its startup time. These constraints also enable aggressive ahead-of-time compilation, thereby reducing the image's time to peak performance.

Static images are not for everyone, due to the closed-world constraint, nor are they for every type of application. They often require manual configuration in order to achieve the best results. We do, however, expect the results to be worthwhile in important deployment scenarios such as small embedded devices and the cloud.

… address … low startup time, slow time to peak performance, and large footprint

… _static images_ to the Java Platform

https://mail.openjdk.java.net/pipermail/discuss/2020-April/005429.html



Project Leyden: Beginnings

mark.reinhold at oracle.com [mark.reinhold at oracle.com]
Fri May 20 14:42:02 UTC 2022

- Previous message (by thread): Welcome to Project Leyden!
- Next message (by thread): Project Leyden: Beginnings
- **Messages sorted by:** [date] [thread] [subject] [author]

The ultimate goal of this Project, as stated in the Call for Discussion [1], is to address the long-term pain points of Java's slow startup time, slow time to peak performance, and large footprint.

....

We will explore a spectrum of constraints, weaker than the closed-world constraint, and discover what optimizations they enable. The resulting optimizations will almost certainly be weaker than those enabled by the closed-world constraint. Because the constraints are weaker, however, the optimizations will likely be applicable to a broader range of existing code -- thus they will be more useful to more developers.

We will work incrementally along this spectrum of constraints, starting small and simple so that we can develop a firm understanding of the changes required to the Java Platform Specification. Along the way we will strive, of course, to preserve Java's core values of readability, compatibility, and generality.

We will lean heavily on existing components of the JDK including the HotSpot JVM, the C2 compiler, application class-data sharing (CDS), and the `jlink` linking tool.

In the long run we will likely embrace the full closed-world constraint in order to produce fully-static images. Between now and then, however, we will develop and deliver incremental improvements which developers can use sooner rather than later.

Let us begin!

- Mark

https://mail.openjdk.org/pipermail/leyden-dev/2022-May/000001.html

We will explore a spectrum of constraints, weaker than the closed-world constraint, and discover what optimizations they enable.

... applicable to a broader range of existing code -- thus they will be more useful to more developers.

Leyden: Need source changes to say what we mean

GraalVM™ CRIU OpenJDK

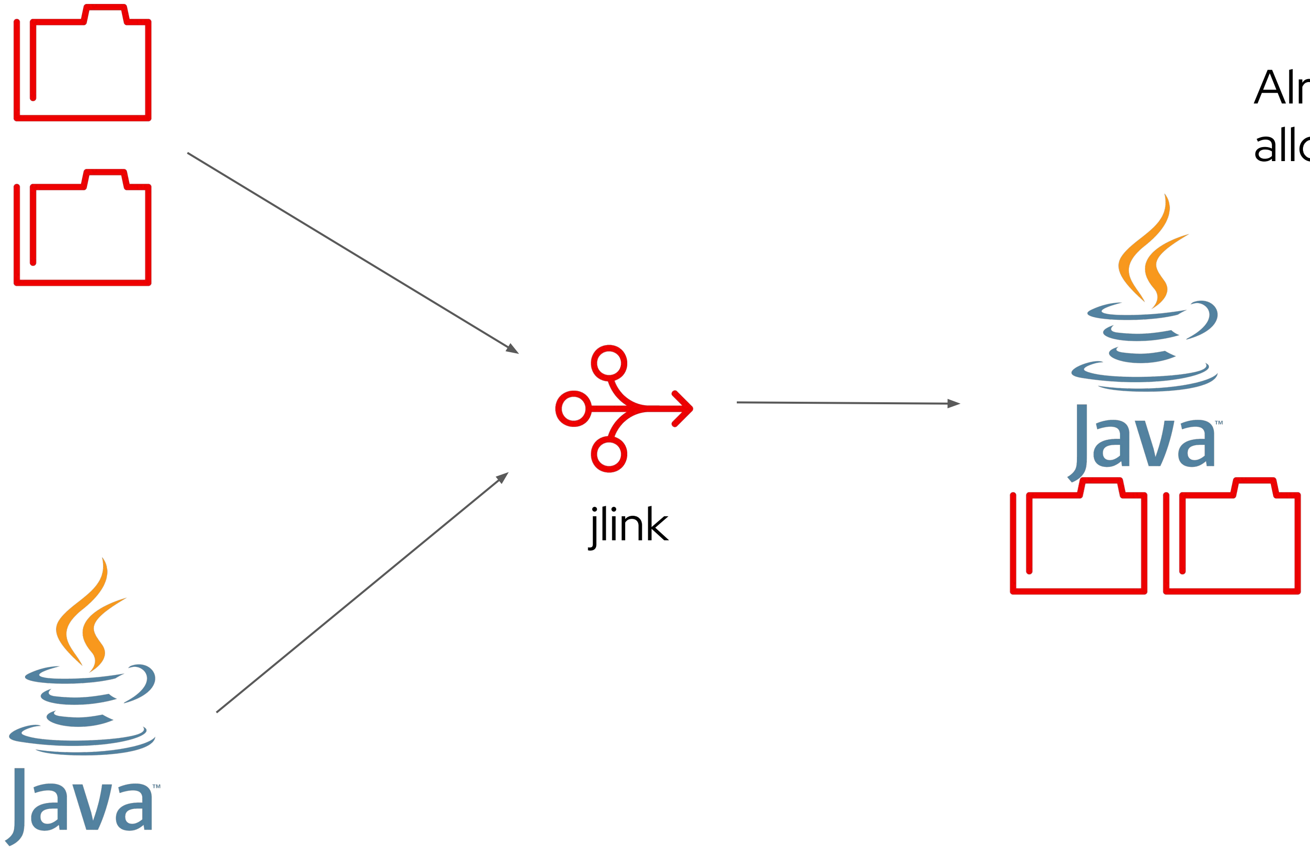
- Every solution has required changes to the code to enable fast startup
  - OpenJDK: Lazy initialization: write Init-On-Demand-Holder pattern
  - Native Image: Closed world constraint and related consequences
  - CRIU: Lifecycle API, portability changes

- Fundamental truth: old code + new semantics => errors!

  (or at least change the program's meaning)

- **Java Language changes are needed!**
  - **One way to say when something should be initialized**

Leyden: need a tool to apply constraints



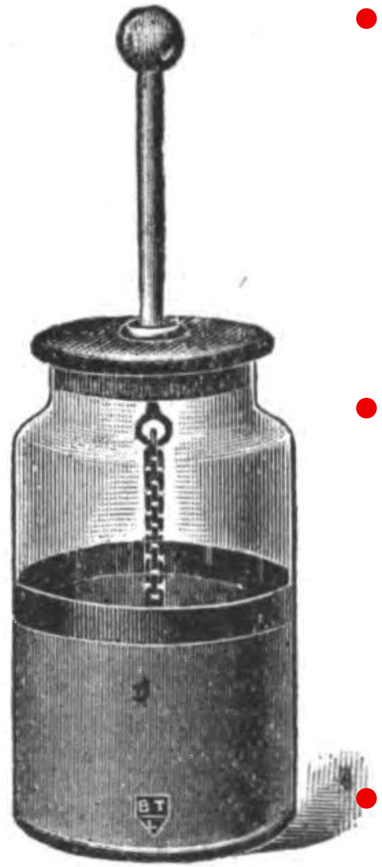
Jlink generates a customized runtime given a JVM and a set of modules.

Already has a plugin architecture that allows modifying Classes

Leyden: Jlink experiments

- Ex.1: Pre-generate Lambda classes during jlink process
  - Lots of user visible changes from this!
    - NestMate / NestHost changes for both generated classes and their hosts
    - Class names change - from Foo$1/0x0000000800c019f0 -> Foo$1
    - Lambda classes are no longer hidden anon classes
    - Class.forName can find them
    - Timing of class loads
    - Stack traces
    - .....
- Ex.2: Convert Class.forName -> ldc
  - Exception blocks
  - Class initialization
- **Java specification changes needed!**
  - **Need to know what changes are valid according to the spec**

Leyden: Requirements



- Leyden needs to give us:
  - Language changes to say what we mean
  - Specification changes about what can validly change
  - A tool to apply the “spectrum of constraints” and generate Leyden images

- And some way to generate the three essentials from language+spec+tool:
  - Cached Class metadata
  - Heap archives
  - AOT compiled code

- That all translates into improvements in startup time!

Wrap up

- Determine how important startup actually is for your workloads

- Pick the option that best matches your use case: JVM, CRIU or Native Image
  - Beware the tradeoffs between throughput / startup / footprint
  - Operationalize it!
  - Share your experience on the OpenJDK Leyden list

- Shift work to build time where possible
  - New style frameworks are great for helping with this!

- Prepare to make changes
  - All solutions require some source changes, Leyden will be no different

Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat