# Understand the trade-offs using compilers for Java applications

## (From AOT to JIT and Beyond!)

Mark Stoodley
Eclipse OpenJ9 & OMR project co-lead
Senior Software Developer @ IBM Canada

mstoodle@ca.ibm.com
@mstoodle

# Java ecosystem has a rich history exploring native code compilation!

- JIT
  - 1999: Hotspot JVM ( https://en.wikipedia.org/wiki/HotSpot ) released by Sun Microsystems
  - 1999: IBM SDK for Java included productized JIT compiler originally built by IBM Tokyo Research Lab, used until Java 5.0
  - 2000: jRockit released by Appeal Virtual Machines (https://en.wikipedia.org/wiki/Appeal_Virtual_Machines)
  - 2006: IBM SDK for Java 5.0 includes J9 JVM with “Testarossa” JIT, now open source as Eclipse OpenJ9
  - 2017: Azul released Falcon JIT based on LLVM
  - 2018: Graal compiler available as experimental high opt compiler in Java 10
- AOT
  - 1997: IBM High Performance Compiler for Java (https://link.springer.com/chapter/10.1007/978-1-4615-4873-7_60)
    - Statically compiled Java primarily for scientific/high performance computing on mainframes
  - 1998: GNU Compiler for Java (gcj) (https://en.wikipedia.org/wiki/GNU_Compiler_for_Java)
    - Statically compile Java used GCC compiler project
  - 2000: Excelsior JET (https://en.wikipedia.org/wiki/Excelsior_JET)
    - Commercial AOT compiler
  - 2017: Experimental jaotc compiler available in OpenJDK9 uses Graal compiler
  - 2018: GraalVM project introduces native images supporting a subset of Java on SubstrateVM
- “Caching” JIT code
  - 2003: jRockit JIT introduces experimental support for cached (but not optimized) code generation
    - https://docs.oracle.com/cd/E13188_01/jrockit/docs142/userguide/codecach.html
  - 2007: IBM ”dynamic AOT” production support introduced in IBM SDK for Java 6
  - 2019: Azul Zing introduces “code stashing” as part of ReadyNow

# Native compilers in today's Java ecosystem

- Hotspot JITS
  - C1 "client" and C2 "server" JIT compilers
  - Default a.k.a. reference native compilers used in OpenJDK
- Eclipse OpenJ9's JIT
  - JIT compiler with multiple adaptive optimization levels (cold through scorching)
  - Historically offered Java compliant AOT compilation for embedded and real-time systems
  - Today caches JIT compilations (a.k.a "dynamic AOT") alongside classes in shared classes cache
- Azul Zing's Falcon JIT based on LLVM
  - Alternative "high opt" compiler to C2
  - Can stash JIT compilations to disk and reload in subsequent runs
- Oracle Graal compiler
  - Written in Java
  - Since Java 9: experimental AOT compiler `jaotc`
  - Since Java 10: experimental alternative to C2 JIT compiler
  - Create native images using SubstrateVM (under "closed world" assumption and other limitations)

# Outline

- **Let's compare:**
  - **JIT**
  - **AOT**
  - **Caching JIT code (== both AOT and JIT!)**
- Taking JITs to the cloud
- Wrap Up

# JIT = Just In Time

- JITs compile code at same time program runs
  - Adapt to whatever the program does “this time”
  - Adapt even to the platform the program is running on

- After more than two decades of sustained effort:
  - JIT is the leader for Java application performance
  - Despite multiple significant parallel efforts aimed at AOT performance

- Why is that?  At least 2 reasons you may already know…

# 1. JITs speculate on class hierarchy

- Calls are virtual by specification
  - But many calls only have a single target (monomorphic) in a particular program run
- JITs speculate that this one target will continue to be the only target
  - Optimize aggressively and keep going deeper (calls to calls to calls….)
- Speculation can greatly expand ability to inline call targets
  - Which expands optimization scope
  - Compiling too early, though, can fool compiler to speculate wrongly

# 2. JITs use profile data collected as program runs

- Not all code paths execute as frequently
  - Profile data tells compiler which paths are worth optimizing
- Not all calls have a single possible target
  - Profile data can prioritize to enable method inlining most profitable target(s)
- Efficient substitute for some kinds of larger scope compiler analyses
  - Takes too long to analyze entire scope but low overhead profile data still identifies constants
  - Contributes to practical compile time
  - BUT accumulating good profile data takes time
- JIT compilers work very well if the profile data is high quality

# But JIT performance advantage isn't free

- Collecting profile data is an overhead
  - Cost usually paid while code is interpreted : slows start-up and ramp-up
  - Quality data means profiling for a while: slows ramp-up

- JIT compilers consume transient resources (CPU cycles and memory)
  - From under a millisecond to seconds of compile time, can allocate 100s MBs
  - Cost paid when compiling : slows start-up and ramp-up
  - Takes time to get to "full speed" because there may be 1000s of methods to compile

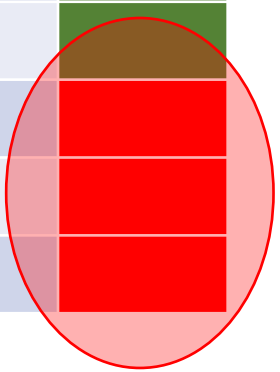
- Also some persistent resource consumption (memory)
  - Profile data, class hierarchy data, runtime assumptions, compiler meta data

# Strengths and Weaknesses

| | JIT |
|---|---|
| Code Performance (steady state) | |
| Runtime: adapt to changes | |
| Ease of use | |
| Platform neutral deployment | |
| Start up (ready to handle load) | |
| Ramp up (until steady state) | |
| Runtime: CPU & Memory | |

# Strengths and Weaknesses

| | JIT |
|---|---|
| Code Performance (steady state) | |
| Runtime: adapt to changes | |
| Ease of use | |
| Platform neutral deployment | |
| Start up (ready to handle load) | |
| Ramp up (until steady state) | |
| Runtime: CPU & Memory | |



**Everyone hopes:**
**Maybe AOT helps here?**

# AOT = Ahead of Time

- Introduce an ”extra” step to generate native code before deploying application
  - *e.g.* run `jaotc` command to convert class files to a platform specific “shared object”
  - Akin to approach taken by less dynamic languages: C, C++, Rust, go, Swift, etc.
  - Still considered “experimental” (JDK9+) and works on x86-64 and AArch64 platforms

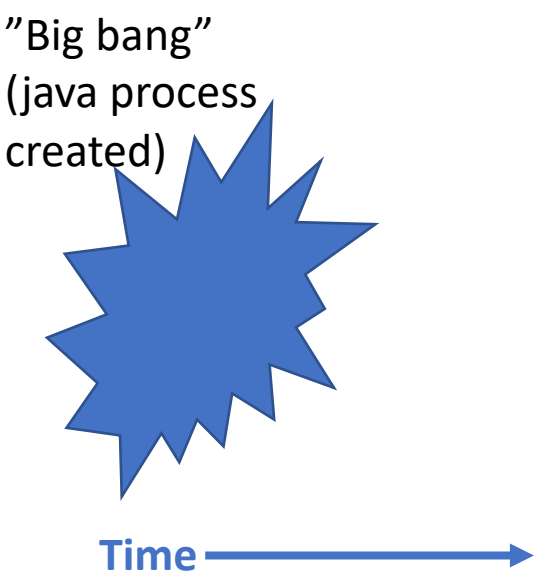
- Two deployment options (decided at build time):
  - No JIT at runtime: statically compiled code runs, anything else interpreted
  - With JIT at runtime: runtime JIT (re)compiles via triggers or heuristics

- AOT has some runtime advantages over a JIT compiler
  - Compiled code performance “immediately” (no wait to compile)
  - Start-up performance can be 20-50% better especially if combined with AppCDS
  - Reduces CPU & memory impact of JIT compiler

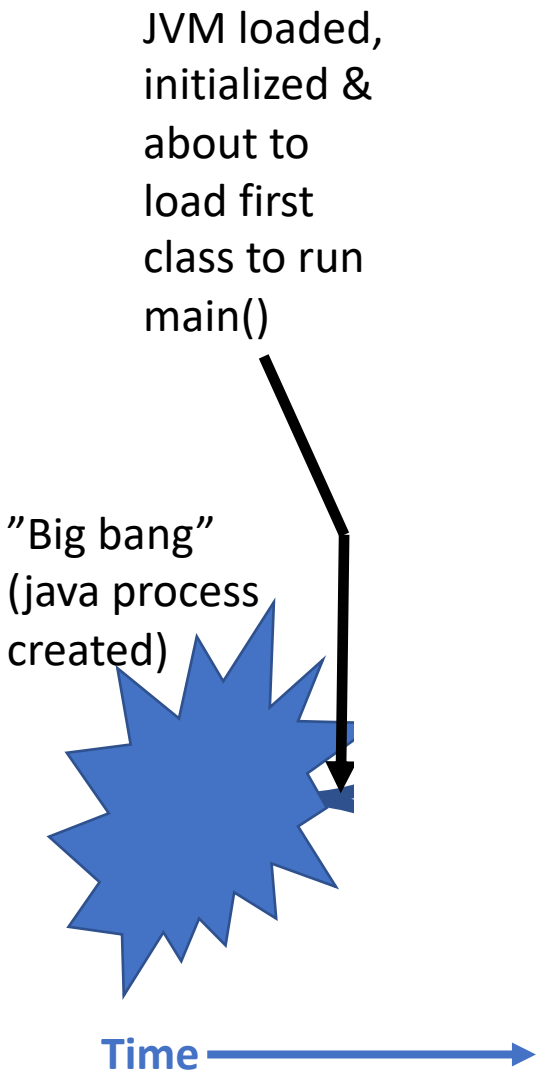
# BUT there are a few big BUTs

- No longer platform neutral
  - Different AOT code needed for each deployment platform (Linux, Mac, Windows)
- Other usability issues
  - Some deployment options decided at build time, e.g. GC policy, ability to re-JIT, etc.
  - Different platforms: different classes load and methods to compile?
  - Ongoing curation for list of classes/modules, methods to compile as your application and its dependencies evolve
  - What about classes that aren't available until the run starts?
- How about those reasons for excellent JIT performance?
  1. Speculate on class hierarchy? Not as easy as for JIT
  2. Profile data? Not as easy as for JIT
- AOT compilers (in pure form) can only reason about what happens at runtime

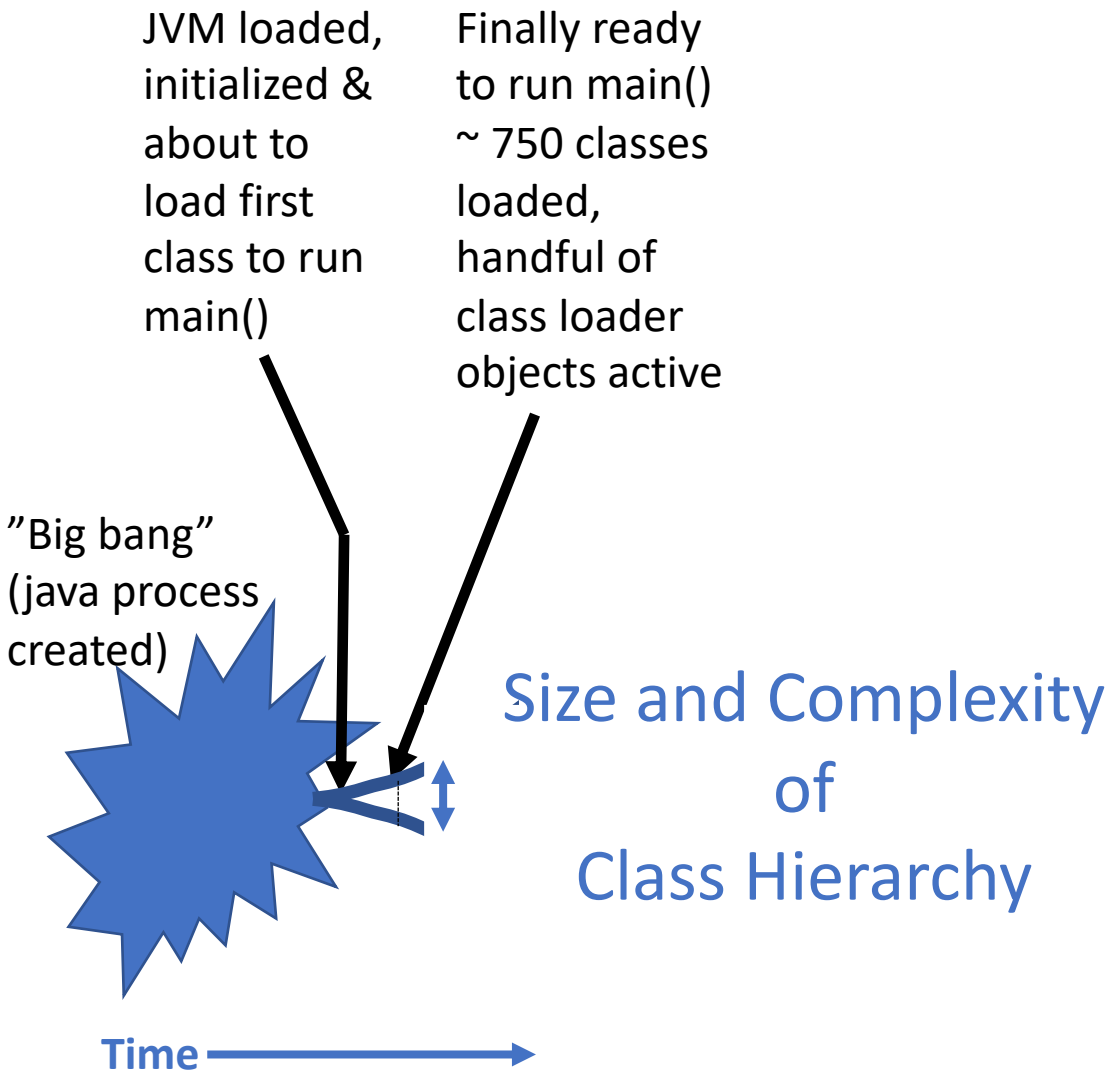
# Sidebar: Life of a running Java application



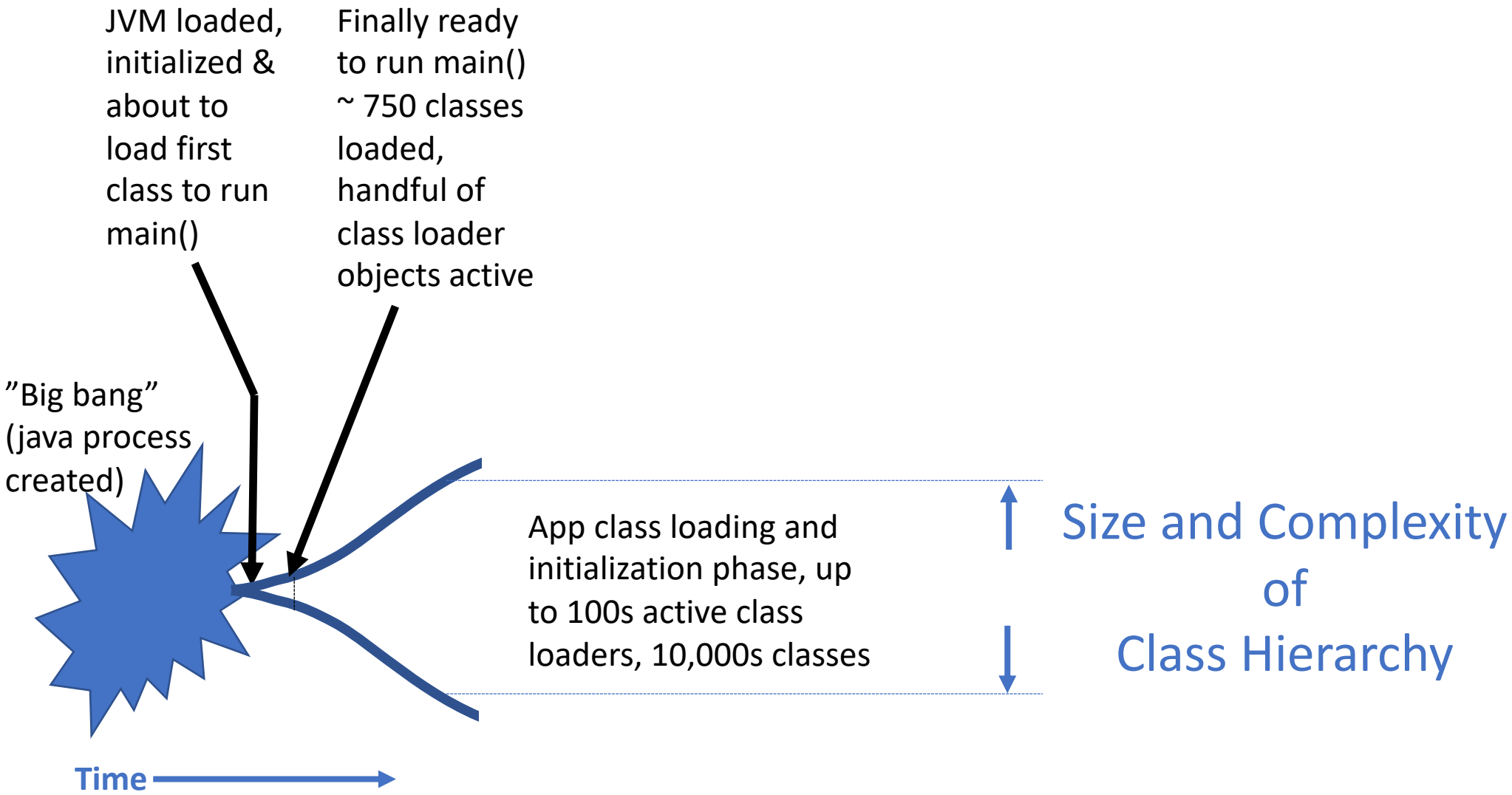
# Sidebar: Life of a running Java application



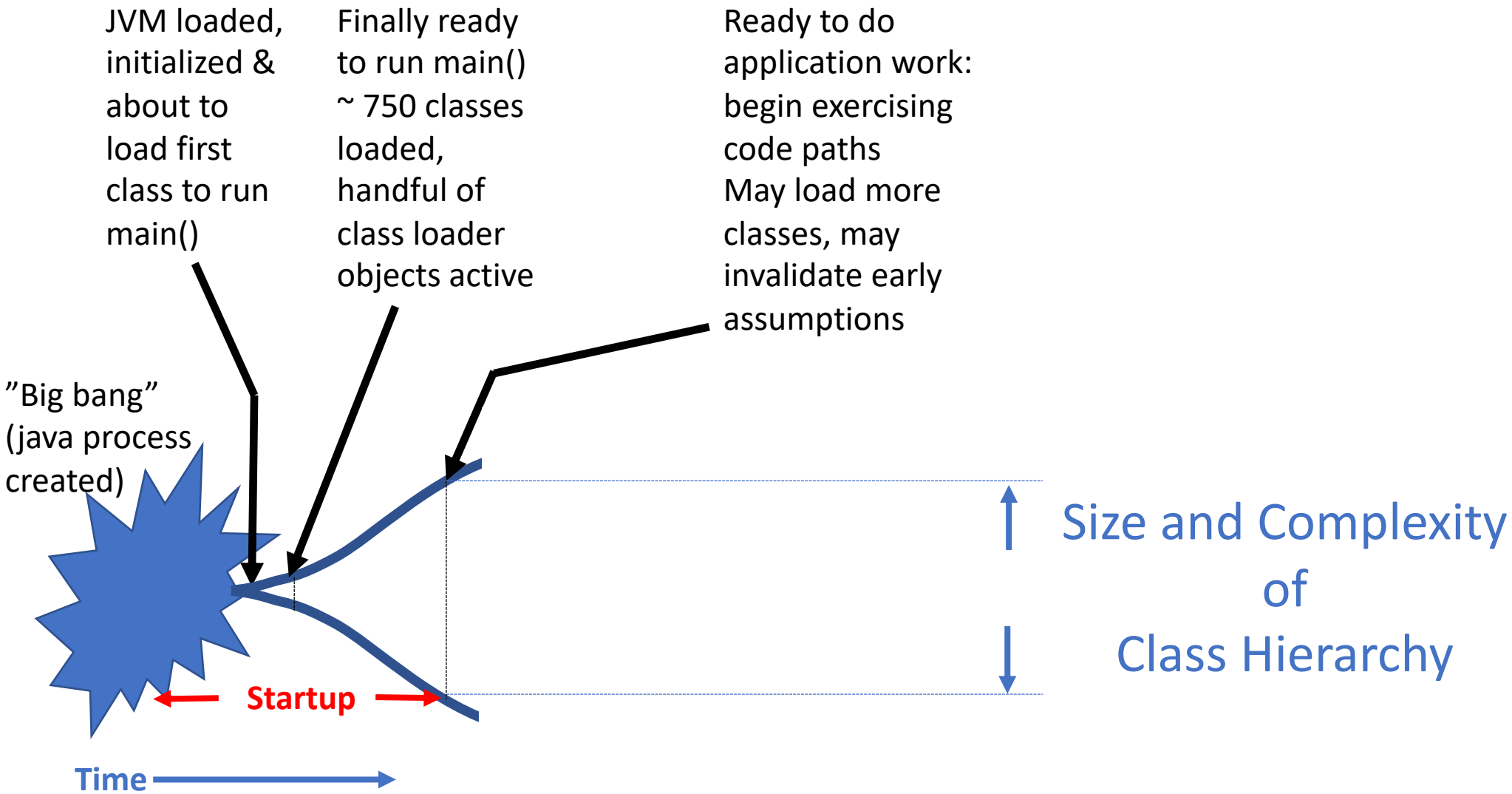
# Sidebar: Life of a running Java application



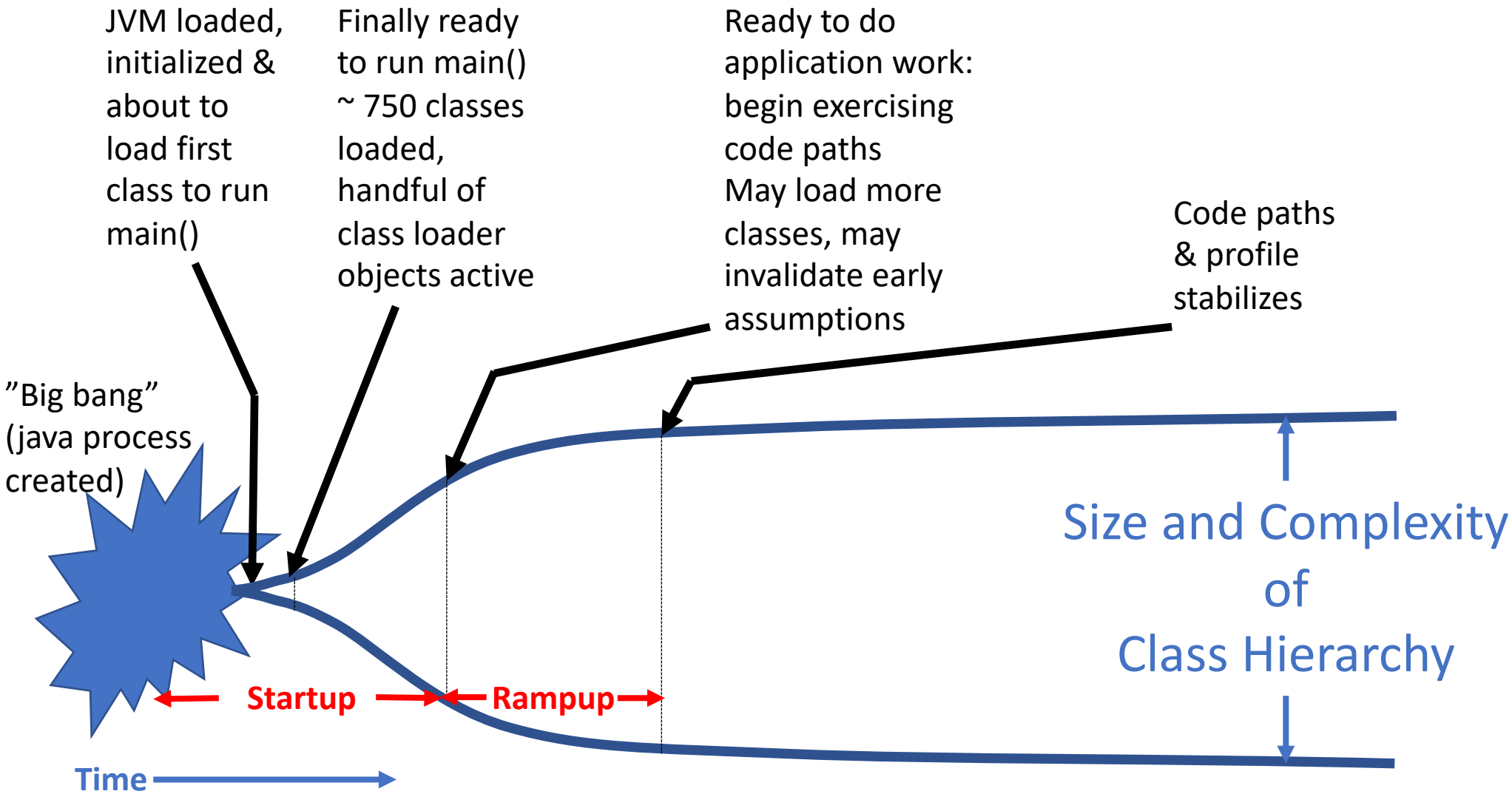
# Sidebar: Life of a running Java application



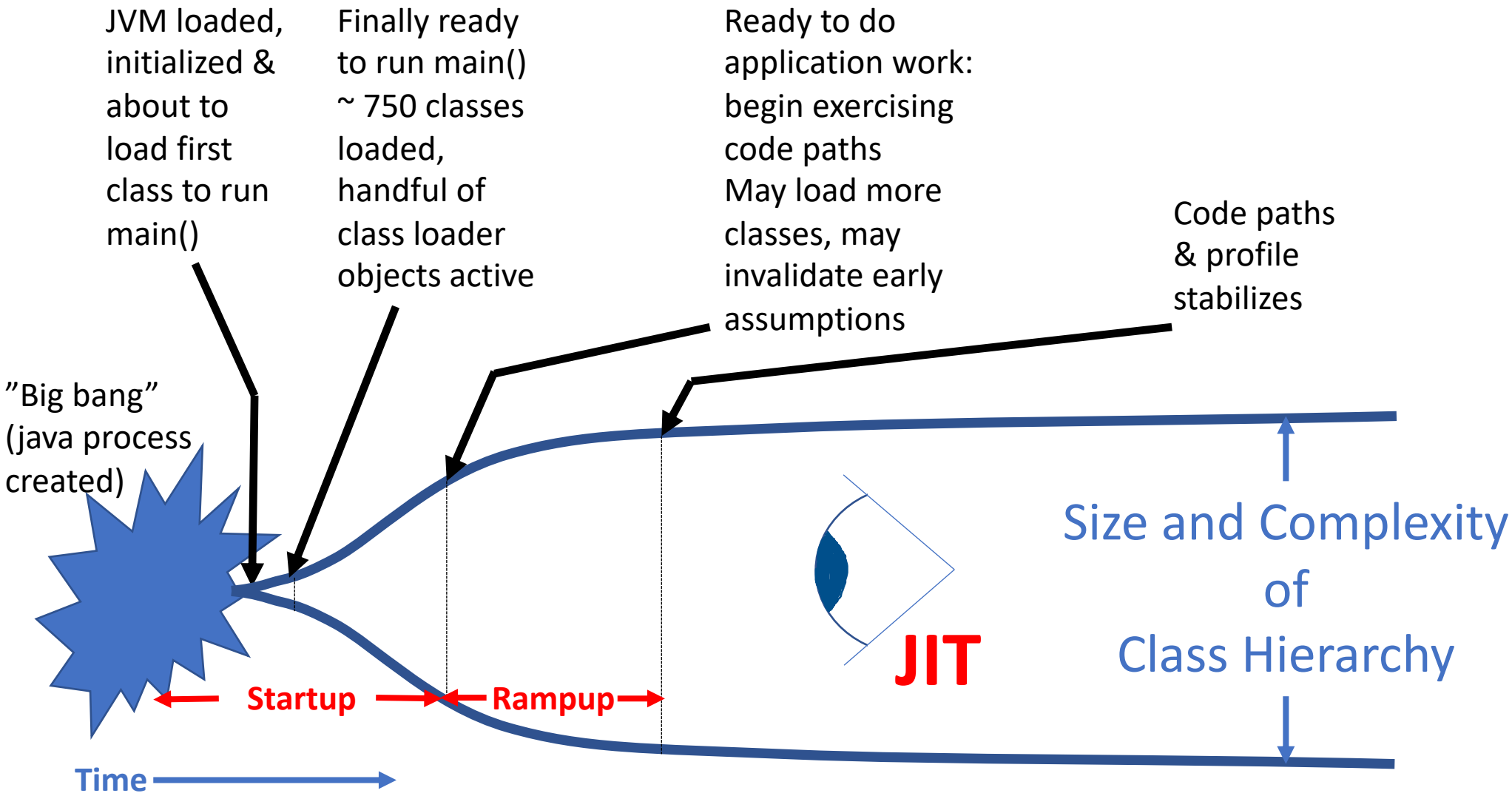
# Sidebar: Life of a running Java application



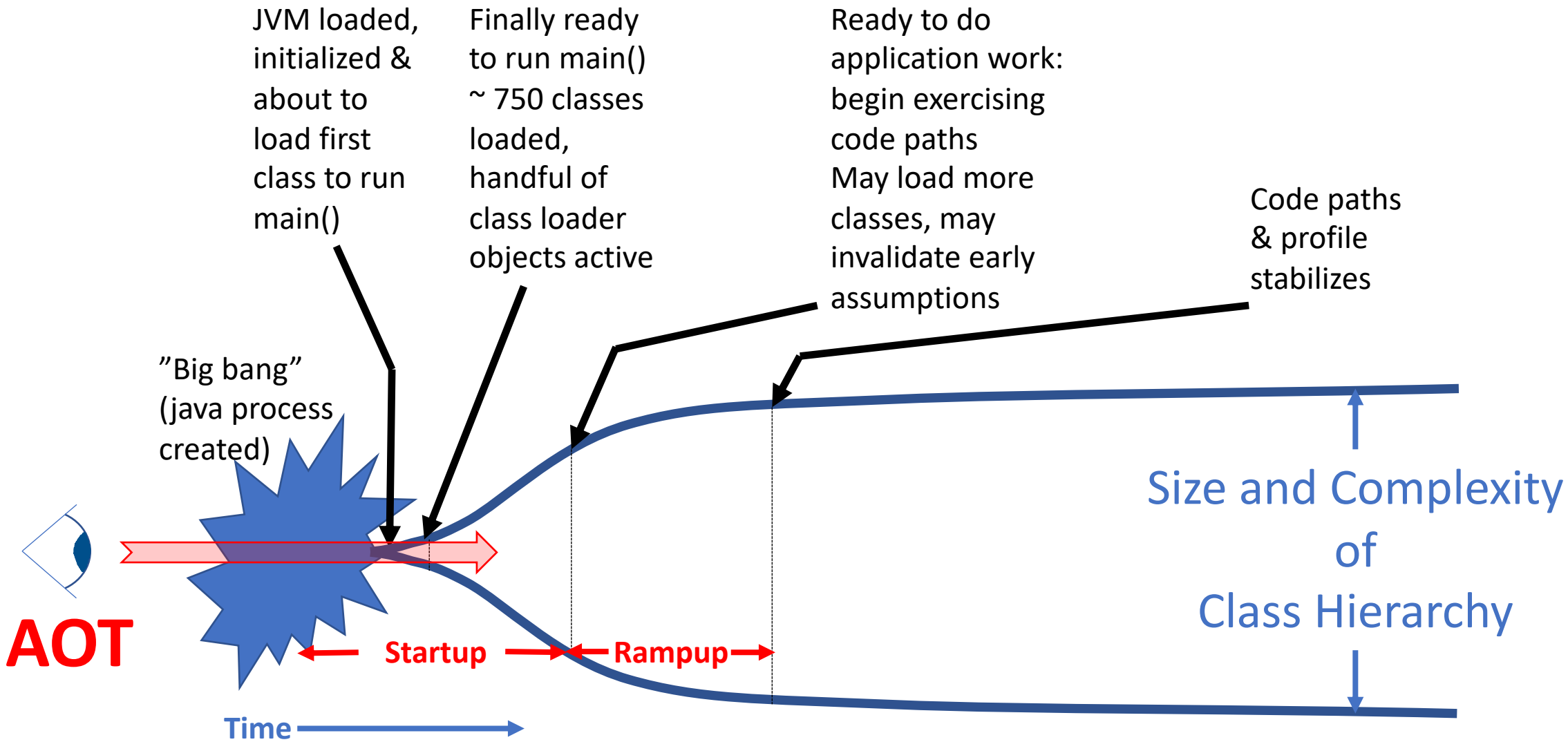
# Sidebar: Life of a running Java application



# JIT compiler's view is inside the process



# AOT compiler's view is through the "big bang"



# So what?

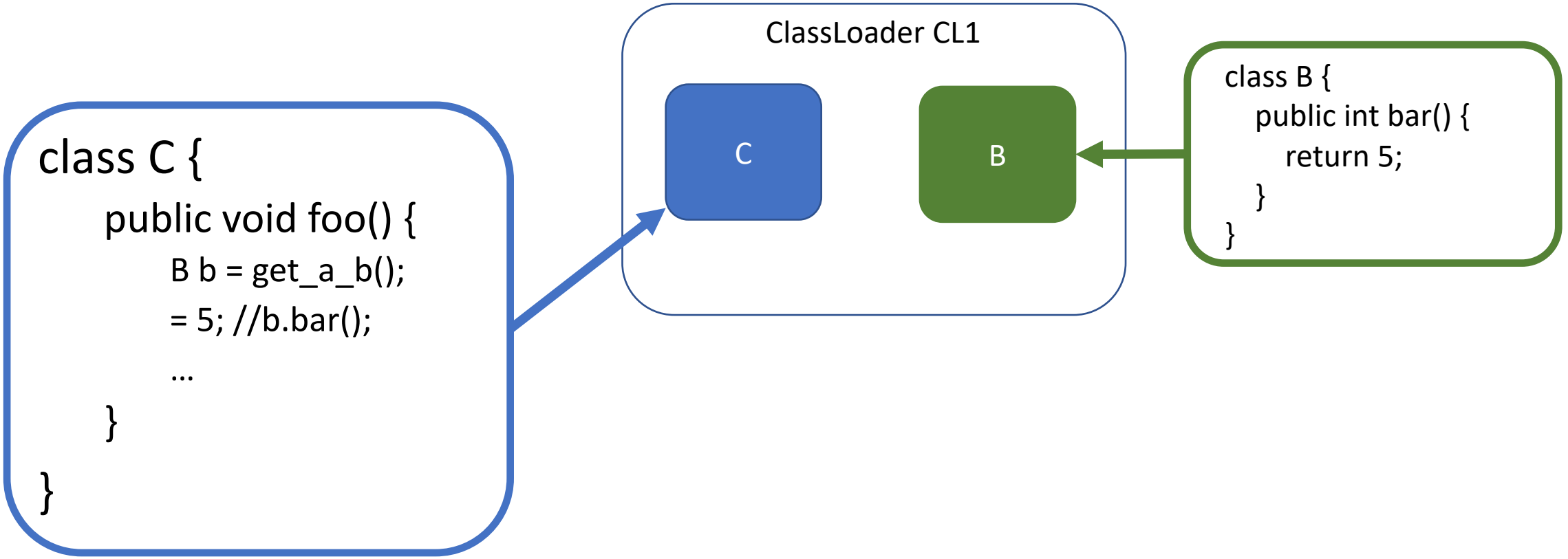
# Imagine two classes B,C: C.foo() calls B.bar() Simple opportunity to inline call to b.bar()?

# Imagine two classes B,C: C.foo() calls B.bar() Can now optimize C.foo() using ‘5’

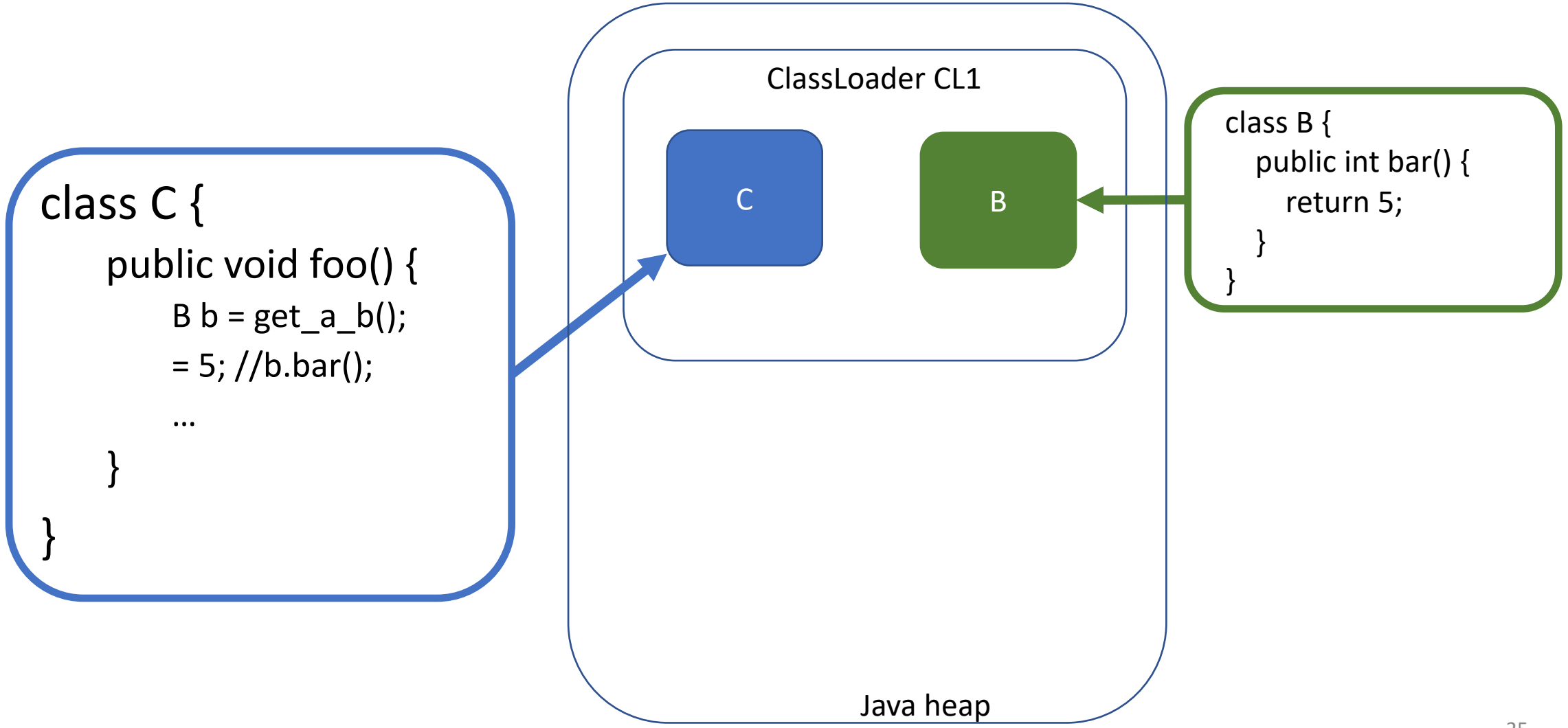
```
class C {
    public void foo() {
        B b = get_a_b();
        = 5; //b.bar();
        …
    }
}
```

```
class B {
    public int bar() {
        return 5;
    }
}
```

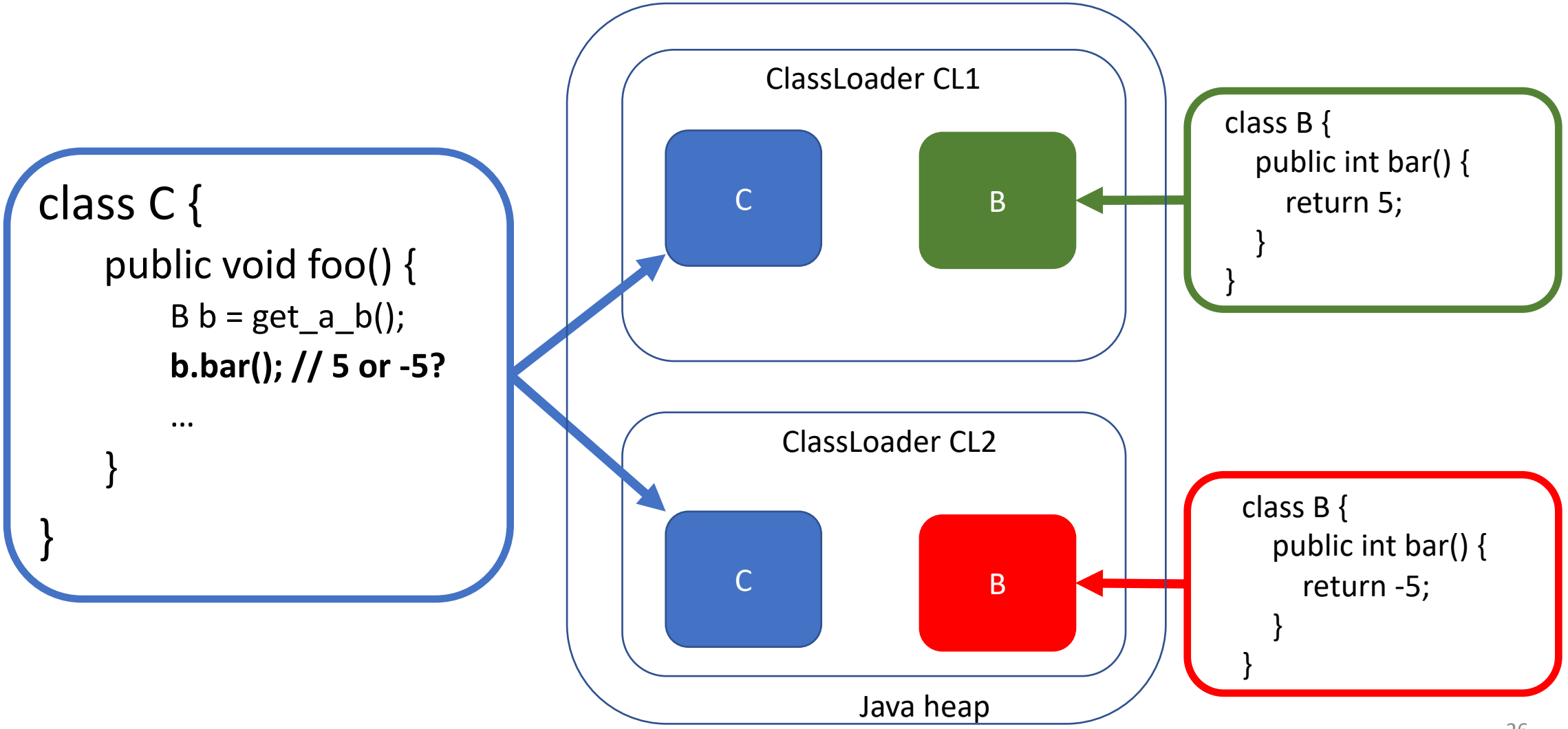
# But C's notion of B is decided by C's class loader



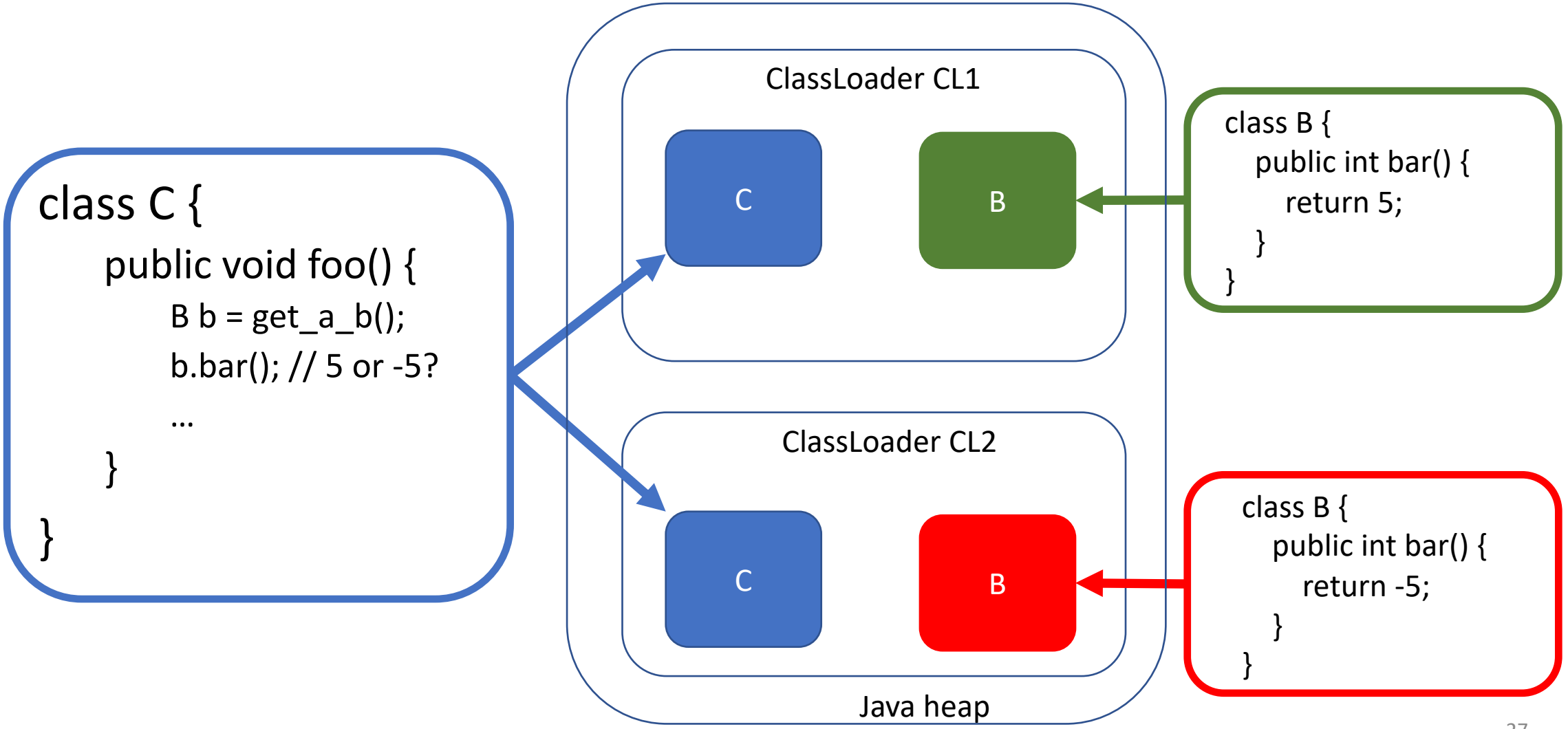
# C's ClassLoader is a Java object created on heap



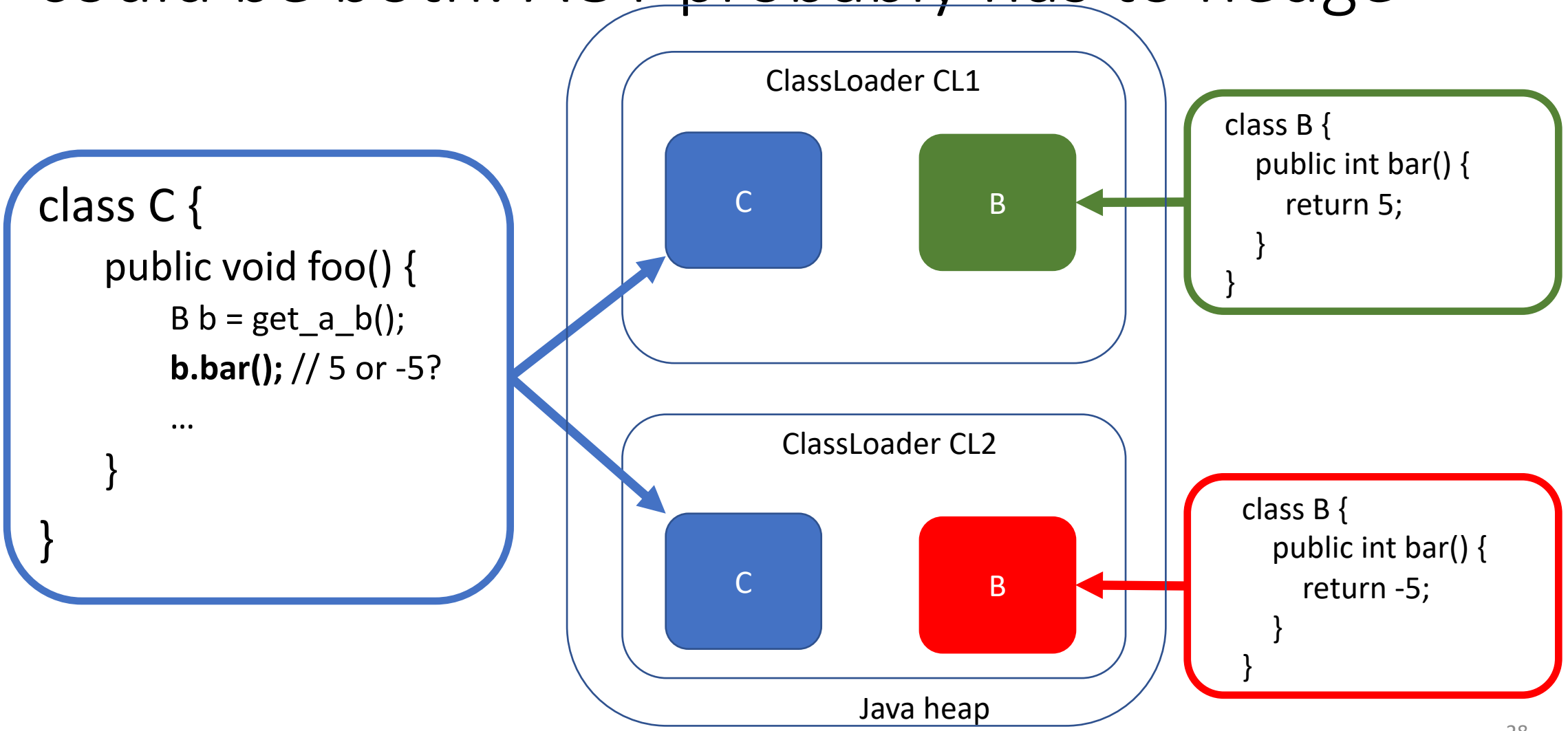
# Class loader objects can invalidate the inlining…



# … and C.foo() may be what resolves B !



# In each run, maybe only CL1 or only CL2 or could be both: AOT probably has to hedge



# Contrived example?

- Modelled on OSGi modules enabling two different versions of the same library to be loaded at the same time (i.e. jar file hell)

- But ask yourself: what *prevents* this scenario if classes can be loaded dynamically and even created on the fly?
  - AOT must completely understand how class loaders will operate at runtime

- JIT acts at runtime and easily deals even with both cases coexisting
  - Each “C” loads as a different `j/l/Class` so each `C.foo()` compiled independently
  - i.e. inline `b.bar()` returning 5 in one case and returning -5 in the other

- For AOT compiler, every inlining hedge reduces optimization scope
  - Increasing gap to JIT performance levels

# Profile Directed Feedback (PDF) may help?

- BUT: AOT code must run **all** possible user executions
  - No longer compiling for “this” user on “this” run
  - Really important to use representative input data when collecting profile for AOT

- Risk: can be misleading to use only a few input data sets
  - AOT compiler can specialize to one data set and then run well on it
  - But PDF can lead compiler astray if data isn’t properly representative

  - Monomorphic in one runtime instance ≠ Monomorphic across all runtime instances

- Benchmarks may not stress AOT compilers properly (not many input sets)
  - Cross training critically important
- Input data sets need to be curated *and maintained* as application and users evolve
  - Profile data collection and curation responsibility is on the application provider
- Observation: PDF has not really been a huge success for static languages

# Strengths and Weaknesses

| | JIT | AOT |
|---|---|---|
| Code Performance (steady state) | | |
| Runtime: adapt to changes | | |
| Ease of use | | |
| Platform neutral deployment | | |
| Start up (ready to handle load) | | |
| Ramp up (until steady state) | | |
| Runtime: CPU & Memory | | |

# Strengths and Weaknesses

| | JIT | AOT | AOT +JIT |
|---|---|---|---|
| Code Performance (steady state) | | | |
| Runtime: adapt to changes | | | |
| Ease of use | | | |
| Platform neutral deployment | | | |
| Start up (ready to handle load) | | | |
| Ramp up (until steady state) | | | |
| Runtime: CPU & Memory | | | |

Is that as good as it gets?

# Caching JIT Compiles

- Basic idea:
  - Store JIT compiled code (JIT) in a cache for loading by other JVMs (“AOT”)
  - Goal: JIT compiled code performance levels earlier
    - Also: reduce JIT compiler’s transient CPU and memory overheads

- Really different than AOT ? No and Yes
  - From perspective of second+ JVM: code loads as if it was AOT compiled
  - First JVM: JIT compiles while app runs but generates code that can be cached
    - Need meta data to validate later runs match first (i.e. same classes loaded same way)
    - If invalid, don’t use cached code: instead do JIT or even more AOT recompilations
- Return to platform neutrality!
  - Different users still get compiled code tailored for their environment

# Two implementations

1. ”Dynamic AOT” in Eclipse OpenJ9 open source JVM[1]
   - Originally introduced in 2007 (IBM SDK for Java 6), currently in JDK8 and later
   - Stores (warm) compiled JIT code to shared memory cache (also persisted on disk)
   - Performance for loaded code within 5%-10% of peak JIT performance (getting better)
   - Resilient to application changes

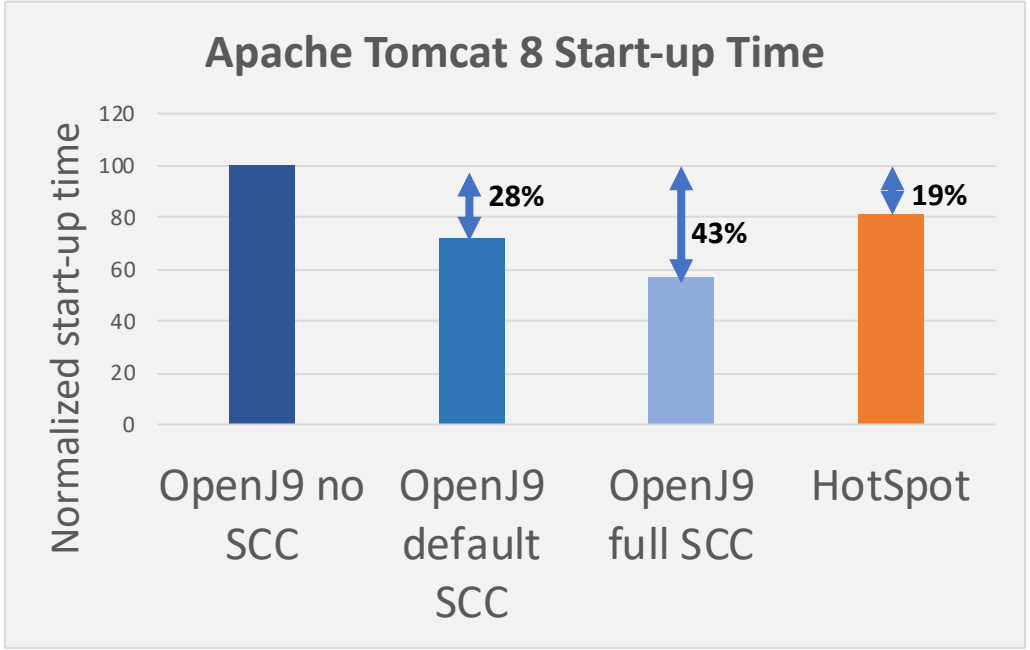
2. “Compile Stashing” in Azul’s proprietary Falcon JIT [2]
   - Introduced in 2018 for JDK8 and later
   - Stores compiled code to disk
   - Stashed code typically reuseable in another run for 60-80% of methods
   - JIT recompilations recover remaining performance
   - Resilient to application changes

[1] https://blog.openj9.org/2018/10/10/intro-to-ahead-of-time-compilation/
[2] https://www.slideshare.net/dougqh/readynow-azuls-unconventional-aot

# OpenJ9: Caching JIT code accelerates start-up

- OpenJ9 Shared Class Cache (SCC)
  - Memory mapped file for caching:
    - Class files*
    - AOT compiled code
    - Profile data, hints
  - Population of the cache happens naturally and transparently at runtime
- Also `-Xtune:virtualized`
  - Caches JIT code even more aggressively to accelerate ramp-up (under load)
  - Maybe slight (5-10%) performance drop

- SCC for JCL bootstrap classes enabled by default
- Use `-Xshareclasses` option for full sharing




* Technically an internal format that can load faster than a .class file

# Strengths and Weaknesses

| | JIT | AOT | AOT +JIT | Cache JIT |
|---|---|---|---|---|
| Code Performance (steady state) | | | | |
| Runtime: adapt to changes | | | | |
| Ease of use | | | | |
| Platform neutral deployment | | | | |
| Start up (ready to handle load) | | | | * |
| Ramp up (until steady state) | | | | * |
| Runtime: CPU & Memory | | | | * |

* After first run

Still some “not green” boxes there

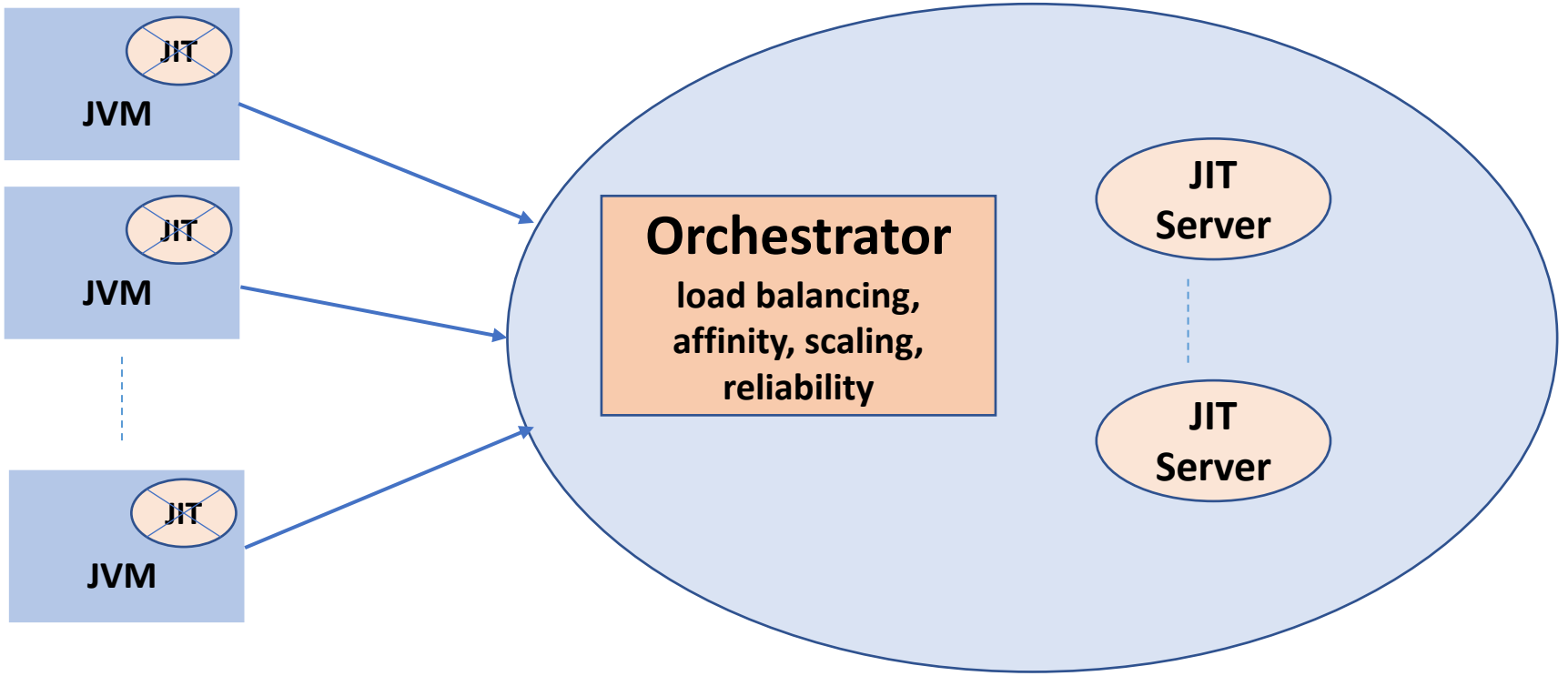
…even for caching JITs…

☹

# Outline

- Let’s compare:
  - JIT
  - AOT
  - Caching JIT code
- **Taking JITs to the cloud**
- Wrap Up

# What if the JIT became a JIT Server



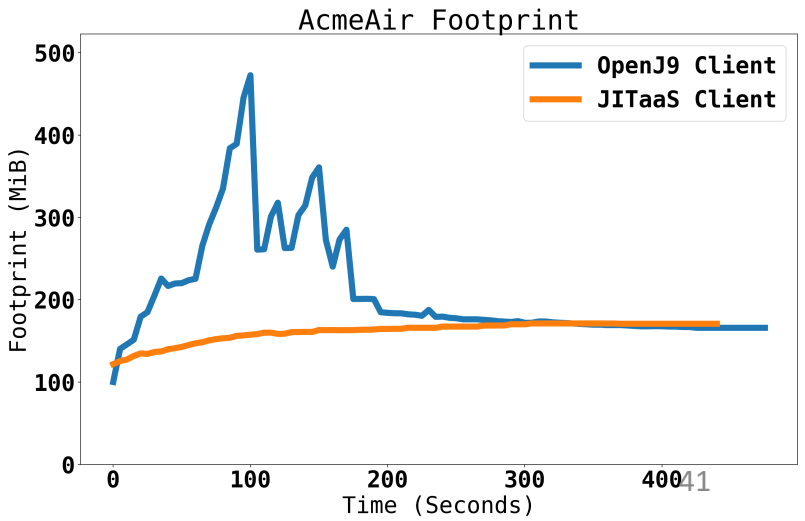

JVM client identifies methods to compile, but asks server to do the actual compilation

- JIT server asks questions to the client JVM (about classes, environment, etc.)
- Sends generated code & meta data back to be installed in client's code cache

# Benefits of an independent JIT server

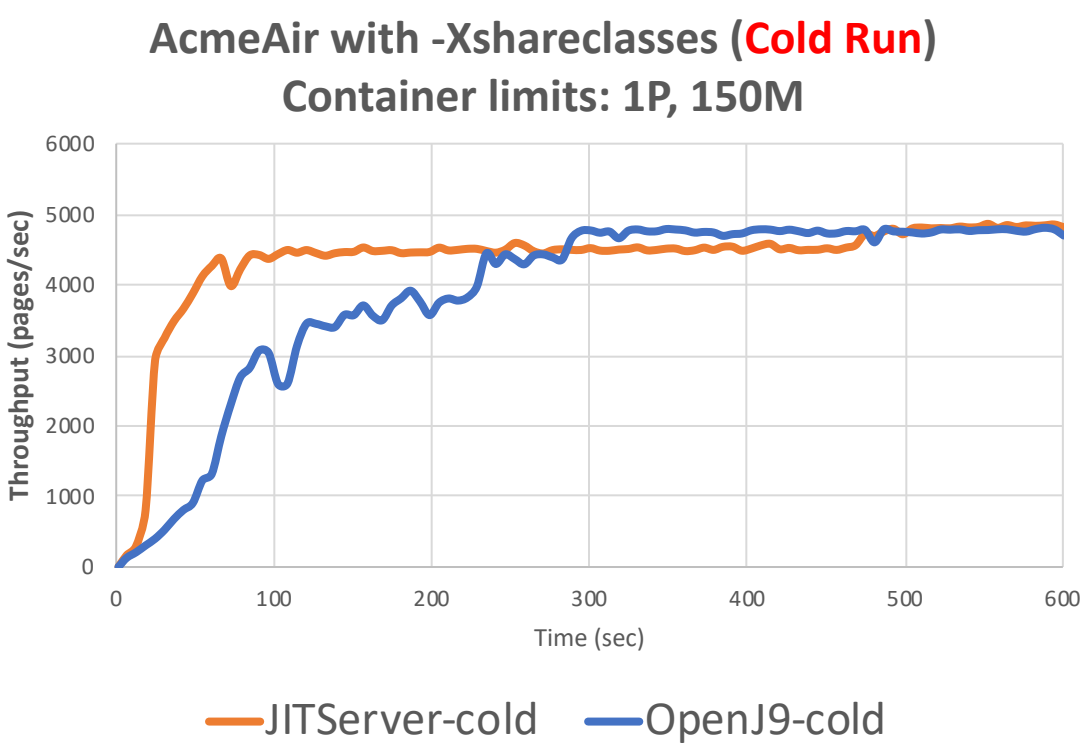
- Move much of JIT induced CPU and memory spikes away from client
  - Client CPU and memory consumption dictated by application

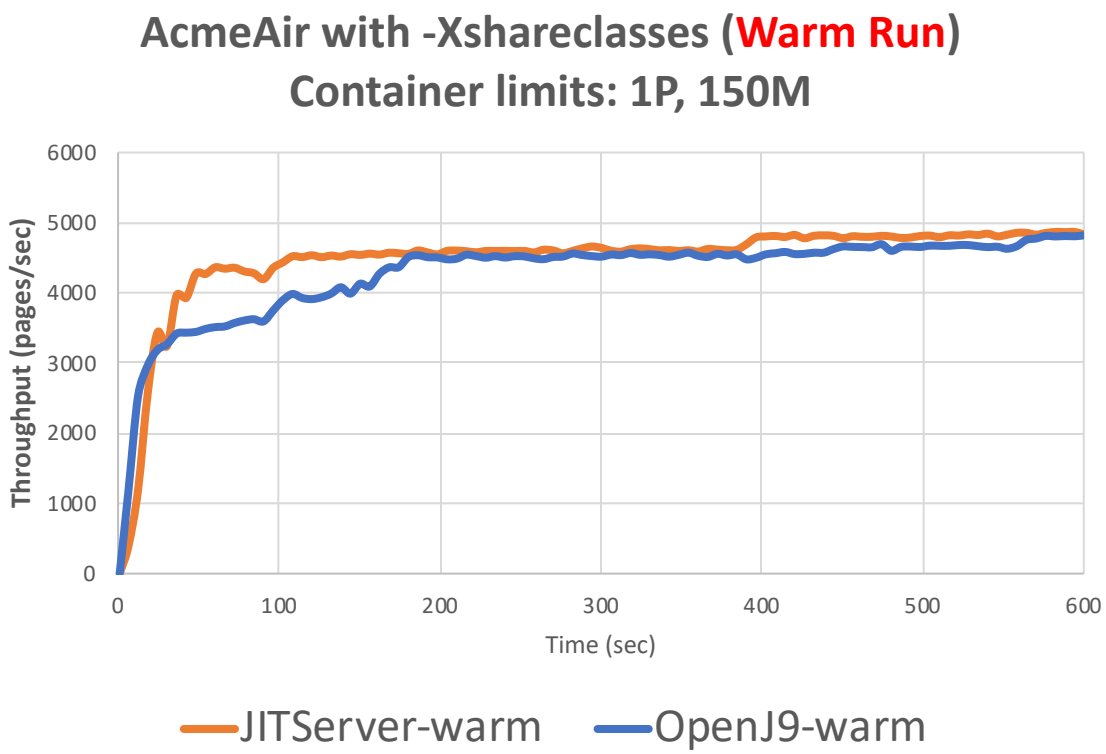
- JIT server connected to client JVM at runtime, so:
  - Theoretically no loss in performance using same profile and class hierarchy info
  - Still adaptable to changing conditions
  - JVM client still platform neutral



# Could that work?

# AcmeAir rampup with JIT Server using `-Xshareclasses`

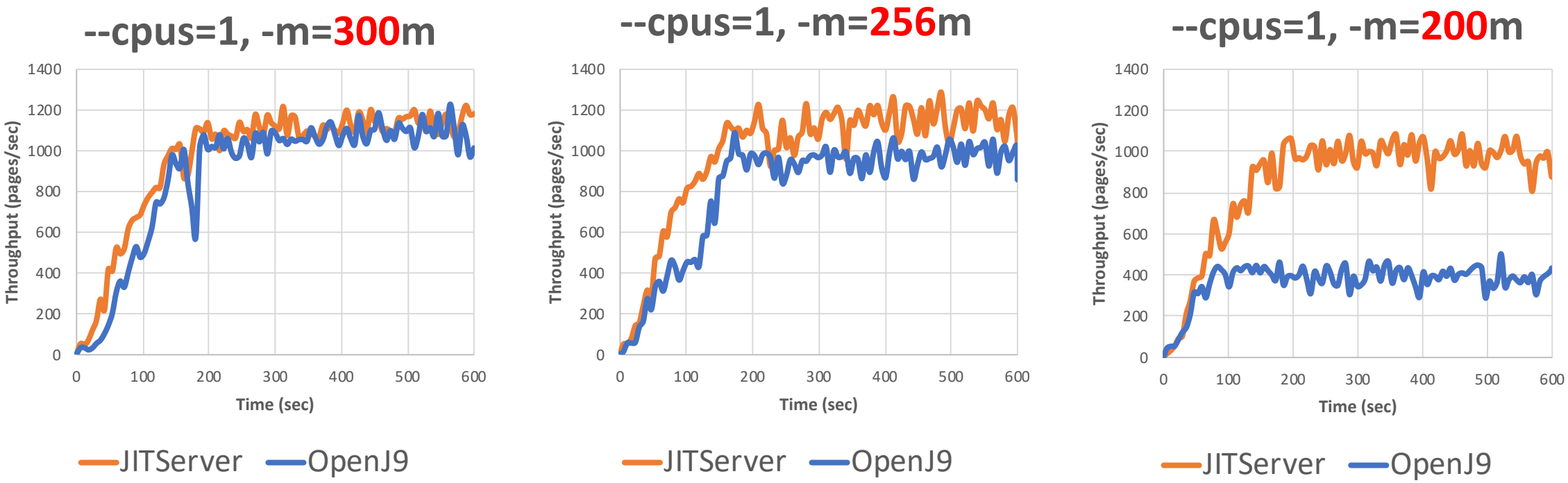



All JVMs run in containers, client and server on different machines with direct cable connection

**Note: Hotspot takes twice as long as OpenJ9 to ramp up to about the same performance level**

# JITServer Performance – Daytrader 7 Throughput




Smaller memory limit

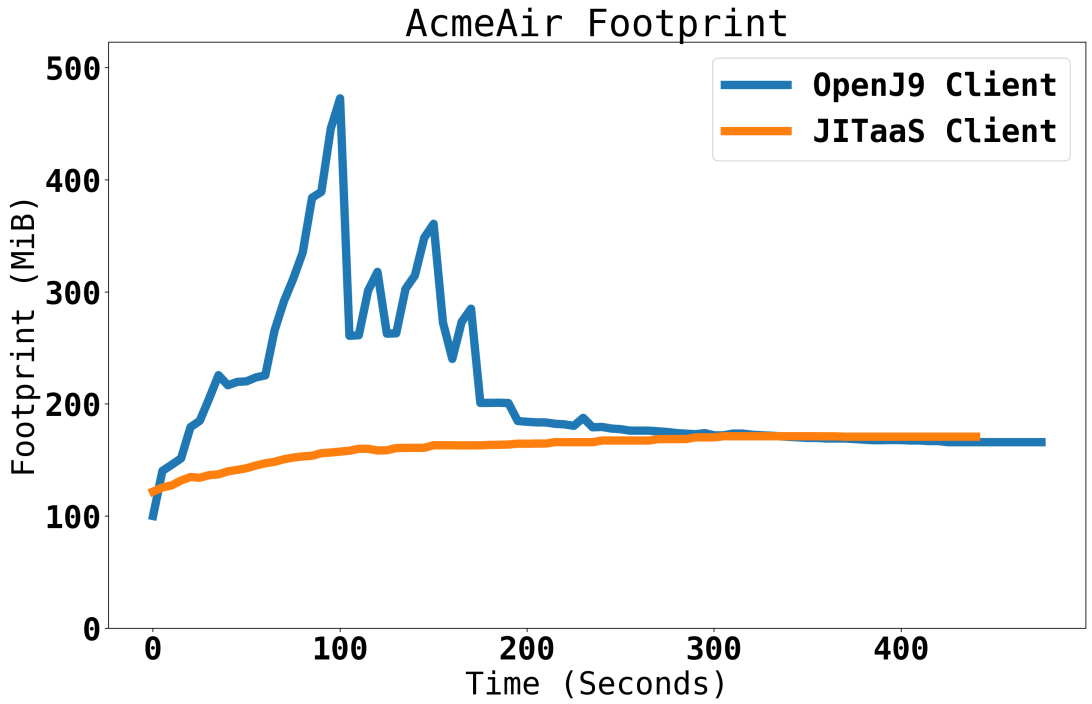
**Throughput benefits grow in constrained environments**

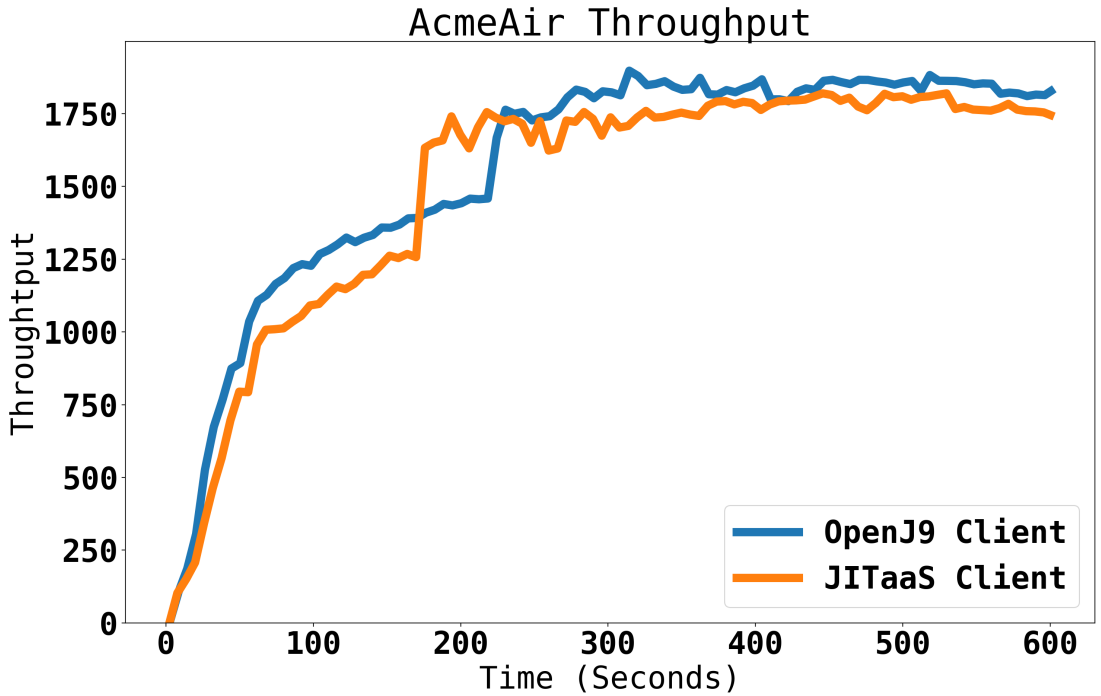
What about network latency?

Won’t that hurt start up and ramp up?

Will it be practical in the cloud?

# JIT Server works well on Amazon AWS!




* JITaaS == JIT Server

# Strengths and Weaknesses

| | JIT | AOT | AOT +JIT | Cache JIT | JIT Server |
|---|---|---|---|---|---|
| Code Performance (steady state) | | | | | |
| Runtime: adapt to changes | | | | | |
| Ease of use | | | | | |
| Platform Neutral deployment | | | | | |
| Start up (ready to handle load) | | | | * | ** |
| Ramp up (until steady state) | | | | * | ** |
| Runtime: CPU & Memory | | | | * | |

* After first run ** After first run across cluster

# JIT Server Current Status

- Code is fully open source at Eclipse Open J9 and Eclipse OMR
  - Has now been merged into our master branch but not yet built-in by default

- Simple options lend well to all kinds of Java workload deployments
  - Server: `java -XX:+StartAsJITServer -XX:JITServerPort=<port>`
  - Client: `java -XX:+UseJITServer -XX:JITServerPort=<port> -XX:JITServerAddress=<host> YourJavaApp`

- Current focus is ensuring stability so it can be built into OpenJ9 by default

- Targeting early 2020 (OpenJ9 0.18 release) to be included in our release binaries (JDK8 and up) at AdoptOpenJDK

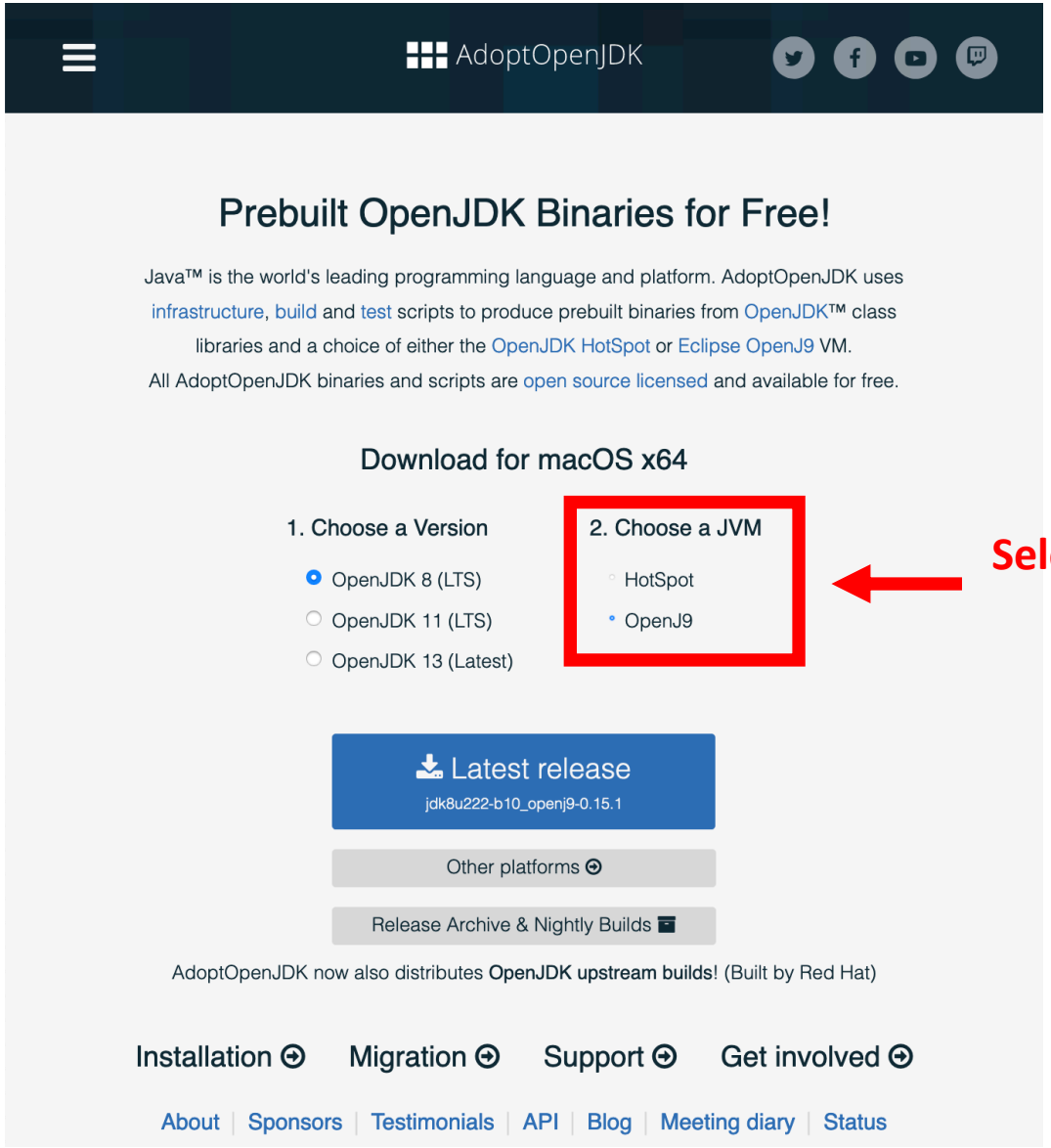
# We are really just at the beginning…

- Primary focus has been on mechanics to move JIT compilation to a server
- Once compilation work is redirected to server :
  - Do that work more efficiently across a cluster of JVMS (think microservices)
  - Classify and categorize JVM clients using machine learning
  - Optimize groups of microservices together
  - …

# Wrapping up

- JITs continue to provide the best peak performance
- AOT compilers can improve start-up by 20-50% but expect steady-state performance to be less than JIT performance
  - Some serious usability issues; I think caching JITs are easier to use
- Caching JIT compilers are within ~5-10% of JIT with excellent start-up and ramp-up even for large complex JakartaEE applications
  - Still room to improve both throughput and start up without sacrificing compliance
- JIT Servers are coming with Eclipse OpenJ9!
  - Hopefully built into AdoptOpenJDK binaries in early 2020!

# https://adoptopenjdk.net




**Select "OpenJ9" Button!!**


Thank You
Suksama
Maake
Asante
Shukria
Dhanyavadagalu
Vinaka
köszönöm
Dankscheen
спасибо
Kiitos
Manana
Dankon
Suksama
Dank Je
Maururu
Biyan
நன்றி
Dziekuje
Chokrane
Juspaxar
Arigato
Gracias
Grazie
Taiku
Diolch i Chi
Gratias Tibi
Terima Kasih
Najis Tuke
Matur Nuwun
감사합니다
धन्यवाद
cảm ơn bạn
Ua Tsaug Rau Koj
Bedankt
Dakujem
Grazas
Tack
Kop Khun Khap
Paldies
Obrigado
ありがとう
Nirringrazzjak
Děkuji
Di Ou Mèsi
Rahmat
谢
Hvala
Хвала
Danke
Merci
Go Raibh Maith Agat
Eskerrik Asko
Salamat
ขอขอบคุณคุณ
Tingki
谢

# Important disclaimers

- THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.
- WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.
- ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.
- ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.
- IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.
- IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.
- NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:
  - CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

# Legal Notice

# Backup

# You can prepopulate Docker containers with Shared Caches (SCCs)

- Prepopulating Docker containers with shared caches very efficient with new ***SCC layers***
  - Working in synergy with Docker layers
- Each Docker layer can prepopulate its own SCC layer that is independent of lower SCC layers
- Each SCC layer can be trimmed-to-fit because upper layers won't add to it
- Layers are **transparent** at runtime
  - Classes and code will load from correct layer
- Significant reduction in disk footprint of Docker images that package a SCC
- Faster pushing/pulling of Docker images from a Docker registry