OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC
Mastodon
Bluesky

Bylaws · Census
Legal

**Workshop**

**JEP Process**

**Source code**
GitHub
Mercurial

**Tools**
Git
jtreg harness

**Groups**
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

**Projects**
(overview, archive)
Amber
Babylon
CRaC
Code Tools
Coin
Common VM Interface
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (..., 23, 24, 25)
JDK Updates
JMC
Jigsaw
Kona
Kulla
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32

# JEP 122: Remove the Permanent Generation

| | |
|---|---|
| *Owner* | Jon Masamitsu |
| *Type* | Feature |
| *Scope* | Implementation |
| *Status* | Closed / Delivered |
| *Release* | 8 |
| *Component* | hotspot / gc |
| *Discussion* | hostspot dash dev at openjdk dot java dot net |
| *Effort* | XL |
| *Duration* | XL |
| *Blocks* | JEP 156: G1 GC: Reduce need for full GCs |
| *Reviewed by* | Paul Hohensee |
| *Endorsed by* | Paul Hohensee |
| *Created* | 2010/08/15 20:00 |
| *Updated* | 2014/08/06 14:14 |
| *Issue* | 8046112 |

## Summary

Remove the permanent generation from the Hotspot JVM and thus the need to tune the size of the permanent generation.

## Non-Goals

Extending Class Data Sharing to application classes. Reducing the memory needed for class metadata. Enabling asynchronous collection of class metadata.

## Success Metrics

Class metadata, interned Strings and class static variables will be moved from the permanent generation to either the Java heap or native memory.

The code for the permanent generation in the Hotspot JVM will be removed.

Application startup and footprint will not regress more than 1% as measured by a yet-to-be-chosen set of benchmarks.

## Motivation

This is part of the JRockit and Hotspot convergence effort. JRockit customers do not need to configure the permanent generation (since JRockit does not have a permanent generation) and are accustomed to not configuring the permanent generation.

## Description

Move part of the contents of the permanent generation in Hotspot to the Java heap and the remainder to native memory.

Hotspot's representation of Java classes (referred to here as class meta-data) is currently stored in a portion of the Java heap referred to as the permanent generation. In addition, interned Strings and class static variables are stored in the permanent generation. The permanent generation is managed by Hotspot and must have enough room for all the class meta-data, interned Strings and class statics used by the Java application. Class metadata and statics are allocated in the permanent generation when a class is loaded and are garbage collected from the permanent generation when the class is unloaded. Interned Strings are also garbage collected when the permanent generation is GC'ed.

The proposed implementation will allocate class meta-data in native memory and move interned Strings and class statics to the Java heap. Hotspot will explicitly allocate and free the native memory for the class meta-data. Allocation of new class meta-data would be limited by the amount of available native memory rather than fixed by the value of -XX:MaxPermSize, whether the default or specified on the command line.

Allocation of native memory for class meta-data will be done in blocks of a size large enough to fit multiple pieces of class meta-data. Each block will be associated with a class loader and all class meta-data loaded by that class loader will be allocated by Hotspot from the block for that class loader. Additional blocks will be allocated for a class loader as needed. The block sizes will vary depending on the behavior of the application. The sizes will be chosen so as to limit internal and external fragmentation. Freeing the space for the class meta-data would be done when the class loader dies by freeing all the blocks associated with the class loader. Class meta-data will not be moved during the life of the class.

## Alternatives

The goal of removing the need for sizing the permanent generation can be met by having a permanent generation that can grow. There are additional data structures that would have to grow with the permanent generation (such as the card table and block offset table). For an efficient implementation the permanent generation would need to look like one contiguous space with some parts that are not usable.

## Testing

Changes in native memory usage will need to be monitored during testing to look for memory leaks.

## Risks and Assumptions

The scope of the changes to the Hotspot JVM is the primary risk. Also identifying exactly what needs to be changed will likely only be determined during the implementation.

This is a large project that affects all the garbage collectors extensively. Knowledge of the permanent generation and how it works exists in both the runtime and compiler parts of the hotspot JVM. Data structures outside of the garbage collectors will be changed to facilitate the garbage collector's processing of the class meta-data in native memory.

Some parts of the JVM will likely have to be reimplemented as part of this project. As an example class data sharing will be affected and may require reimplementation in whole or in part.

Class redefinition is an area of risk. Redefinition relies on the garbage collection of class meta-data during the collection of the permanent generation (i.e., redefinition does not currently free classes that have been redefined so some means will be necessary to discover when the meta-data for classes that have been redefined can be freed).

Moving interned Strings and class statics to the Java heap may result in an Out-of-memory exception or an increase in the number of GCs. Some adjustment of -Xmx by a user may be needed.

With the UseCompressedOops option pointers to class meta-data (in the permanent generation) can be compressed in the same way as pointers into the Java heap. This yields a significant performance improvement (on the order of a few percent). Pointers to meta-data in native memory will be compressed in a similar manner but with a different implementation. This latter implementation may not be as high performance as compressing the pointers into the Java heap. The requirements of compressing the pointers to meta-data may put an upper limit on the size of the meta-data. For example if the implementation required all meta-data to be allocated below some address (for example below the 4g limit) that would limit the size of the meta-data.

## Dependences

Tools that know about the permanent generation will need to be reimplemented. The serviceability agent, jconsole, Java VisualVM and jhat are examples of tools that will be affected.

## Impact

- Other JDK components: Tools that have knowledge of the permanent generation.
- Compatibility: Command line flags relating to the permanent generation will become obsolete.
- Documentation: References to the permanent generation will need to be removed.