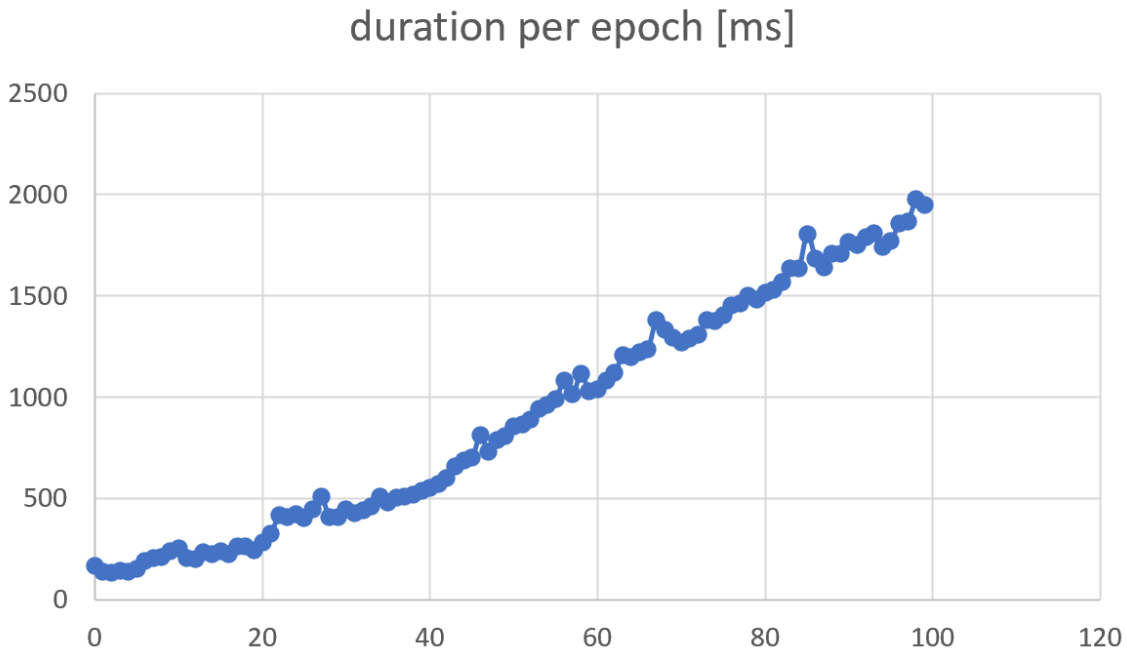
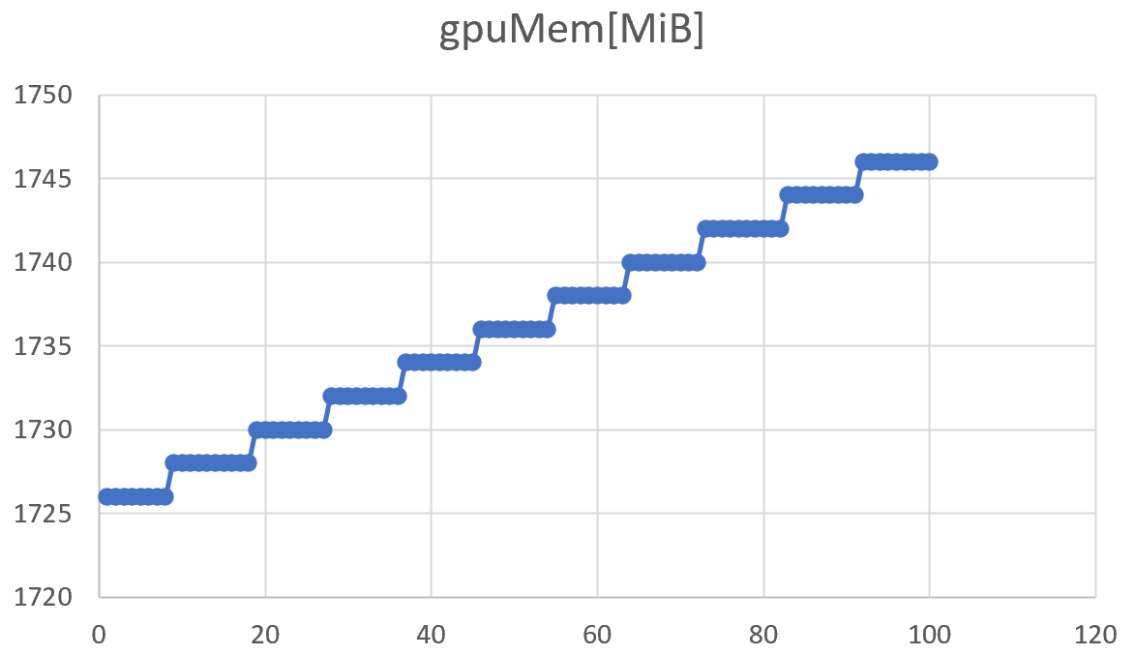
motivation ...

With DJL 0.20.0 I experienced performance problems during training.

Let's illustrate it by measurements on a very basic example from DJL Docs ...





As the duration per epoch increases linearly the integral duration increases quadratically.
The linear growth of the memory on GPU will finally end up with spending all GPU memory and therefore terminating the JVM process.

searching for the cause ...

Searching for the cause revealed:

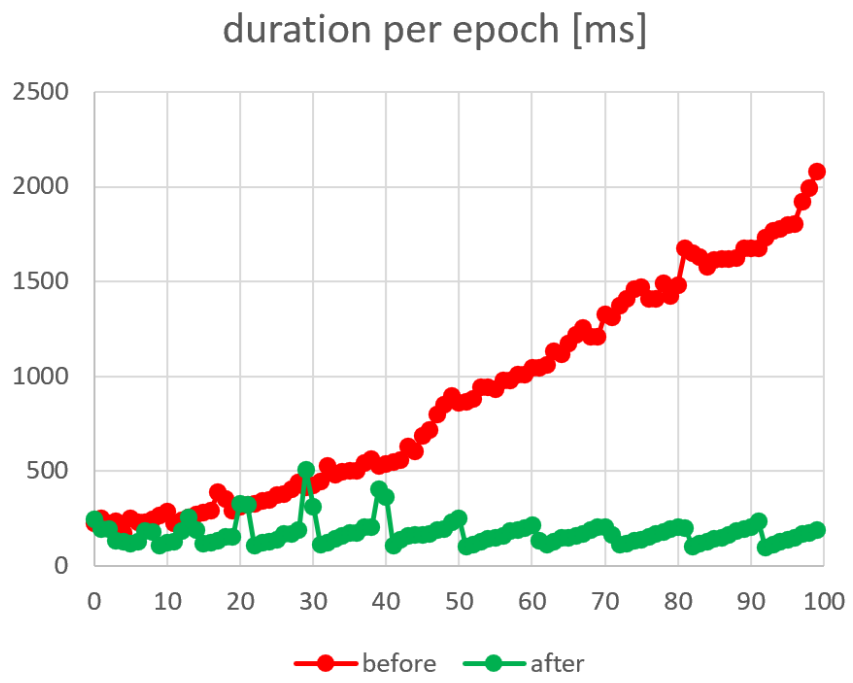
- Unused NDArrays accumulated under the BaseNDManager.
  Linear growth in number over epochs.
- A code snipped was added between DJL 0.19.0 and DJL 0.20.0 to remove gradients on PyTorch engine before each new gradient collection. To do so it was collecting all the NDArrays and thereby translated the memory growth into a duration increase. I see this as a side effect of the growth in the number of unused NDArrays.

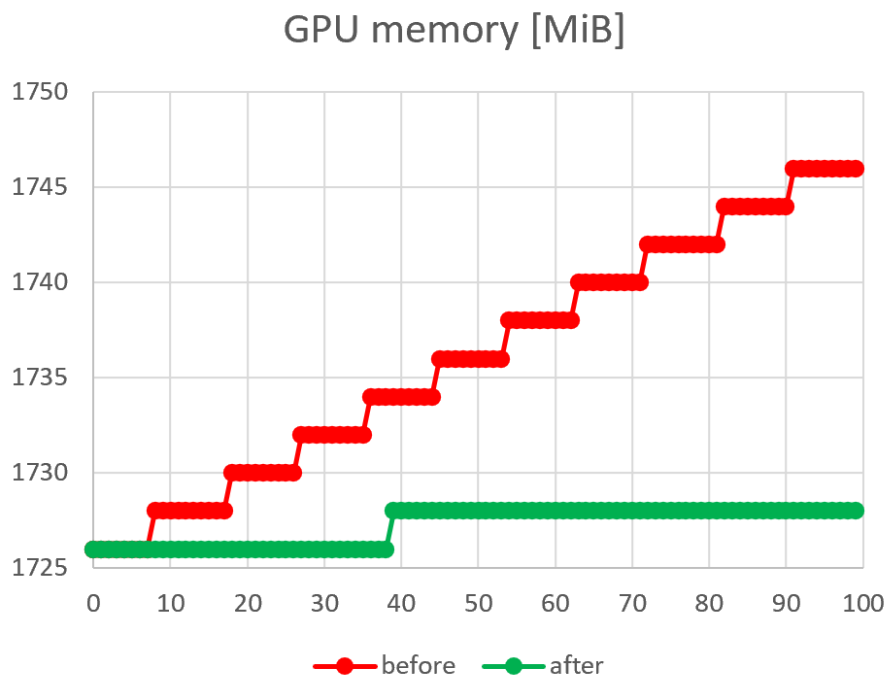
Is this the cause?

To find out ... implement some mechanism to remove the unused NDArrays and measure again.

searching for the cause ... hay-re-ka

With the implemented mechanism to remove the unused NDArrays in place running the same example as before:





The duration and GPU memory situation looks much better.
(Why there still shows up one little memory jump ... an interpretation on the last slide.)

solution approach ... main idea

Use the JVM's gc to

1. identify no more used NDArrays
2. close them

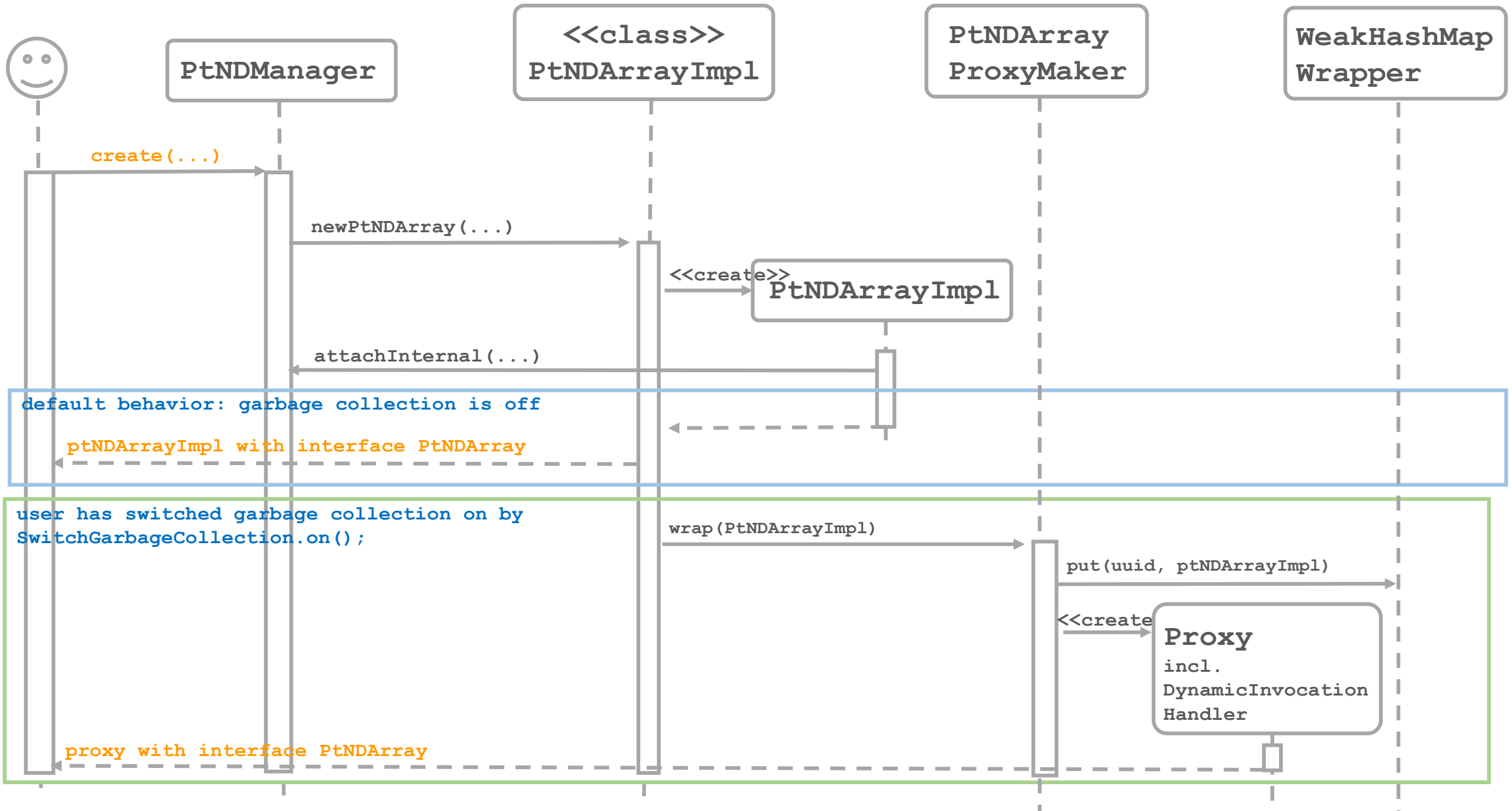
solution approach ... comments on the approach

- it is not the idea to replace the NDManager hierarchy.
- the JVM's gc protects mainly the heap's memory - it does not even care about GPU memory
- the JVM's gc is not deterministically triggerable, its strategy can be choosen at runtime (normally its default behaviour), you don't know at coding time how often it runs and what it does on each run ... but practically it runs often on the time scale our problem piles up
- JVM's gc nicely identifies unused java objects and somewhen cleans them up from the heap
- To remove NDArrays the AutoCloseable mechanism to remove all NDArrays that belong to an NDManager works deterministically. However, there is no mechanism (as far as I know) that garantees in a praticable way that no NDArray that is created within opening and closing of a NDManager will be deleted when leaving that scope if you don't explicitly move it out. Why? Because when using NDArrays from other NDManagers newly created NDArrays may "escape" to that scope if you are e.g. not aware that otherwise commutative operations are not commutative concerning the NDManager the operation result belongs to.
- our approach may have unliked sideeffects - e.g. worse performance in some case. Therefore it should be switched off by default and only switched on at will by the user.

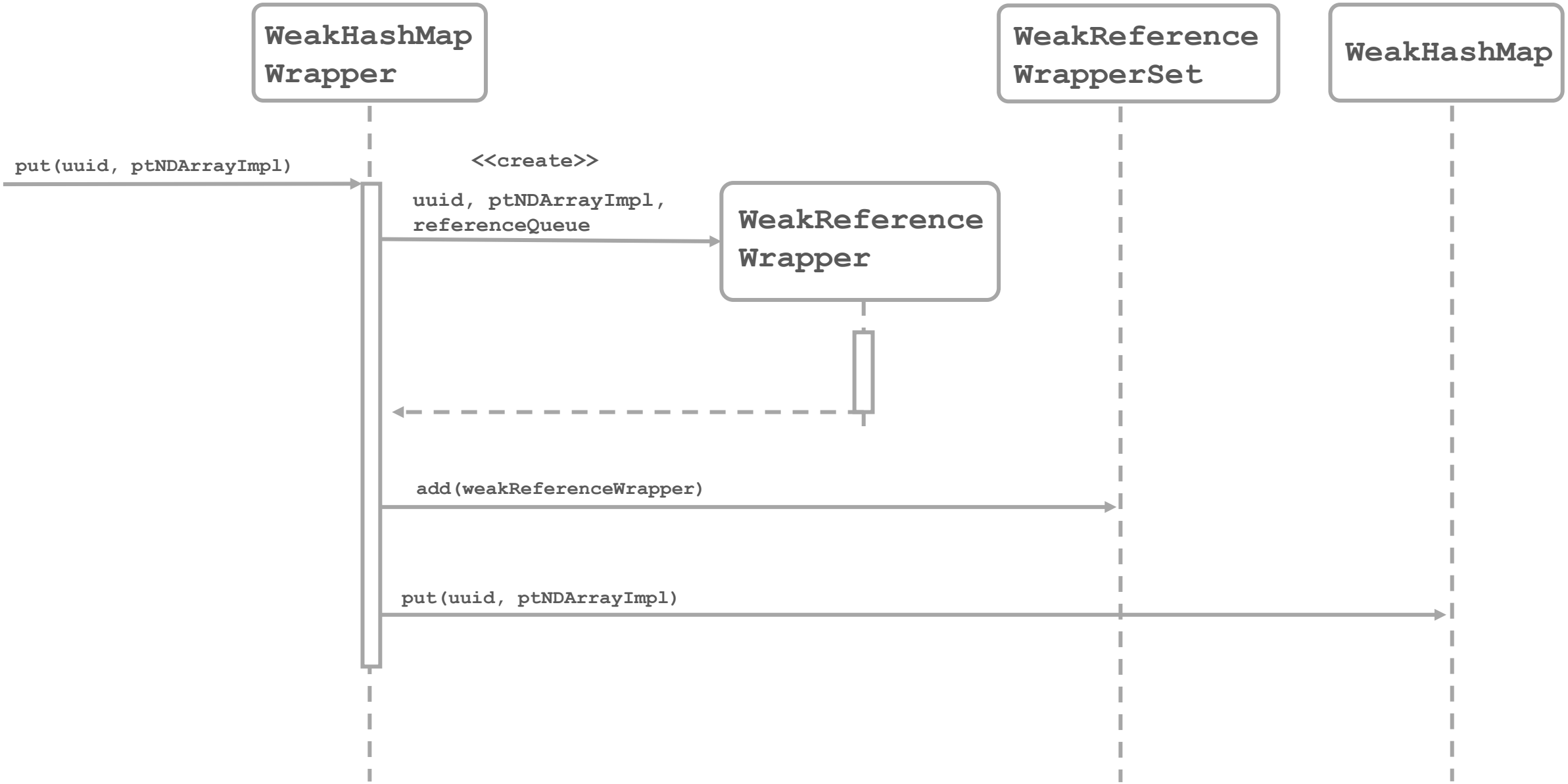
solution approach ... ingredients

- the JVM gc removes java objects that are no more referenced.
  As the NDArrays that are unused by the user are still referenced by their NDManagers we let the users reference only Proxies to the NDArrays.
- We use DynamicProxies. Therefore we have to separate NDArrays into interface and implementation.
- To decouple the DynamicProxies from the NDArrays concerning normal references we store the NDArrays in a WeakHashMap. The Proxies only have an uuid key to retrieve the NDArrays from the WeakHashMap on each method call.
- When the user no longer references a particular DynamicProxy the Proxy and the uuid key get garbage collected (somewhen). On a user call to the method of any of the DynamicProxies (not this particular one) the WeakHashMap clean up the entry (normal behaviour) triggered by polling its ReferenceQueue (on the calling thread - no separate threads here).
- How to hook in here for closing the resource? As the WeakHashMap code is under GPL license we are not touching the code. Instead we facade the WeakHashMap by a wrapper that has its own ReferenceQueue (the ReferenceQueue is the information channel from gc to our code). On a method call to any of the DynamicProxies we first check the ReferenceQueue for newly deleted uuid keys. For any of them we close the actual NDArray before calling the WeakHashMap.

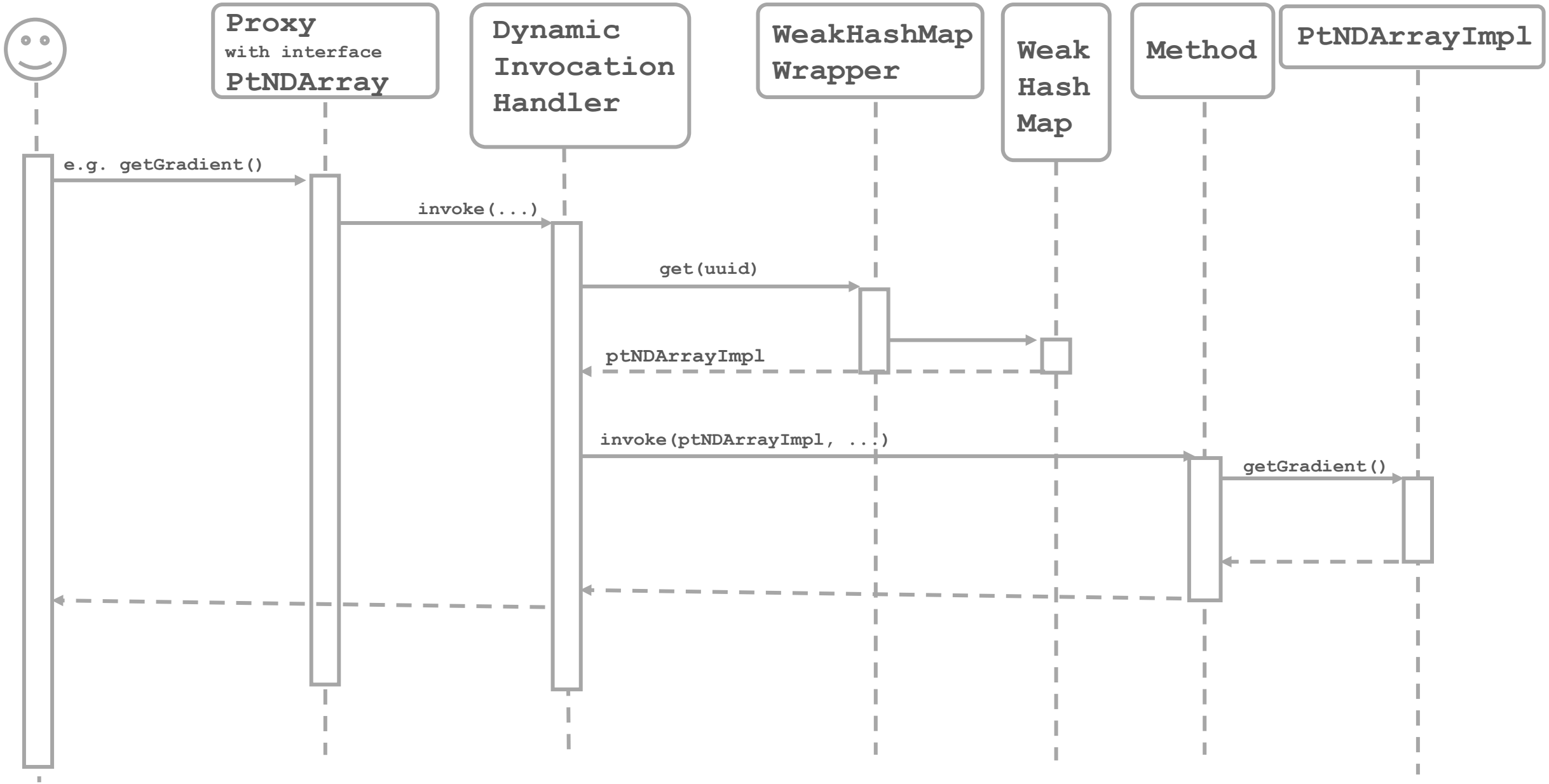
solution approach ... lifecycle: NDArray creation 1



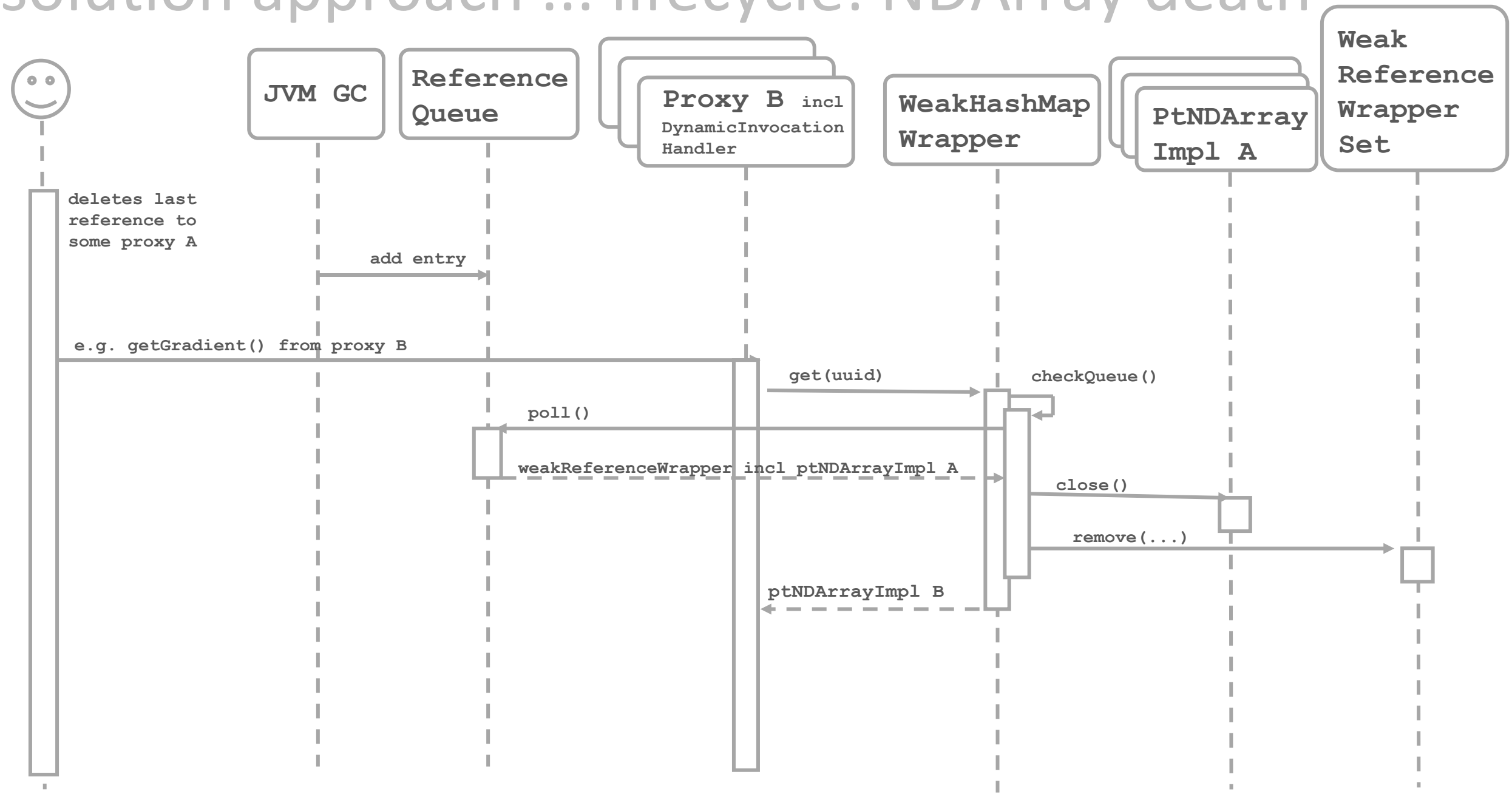
solution approach ... lifecycle: NDArray creation 2



solution approach ... lifecycle: NDArray use



solution approach ... lifecycle: NDArray death



Doc History

Version 2:

- BugFix: removed the weakReferenceWrapper from the corresponding collection
  fixed a heap space memory leak
- Marketing: before/after image

Version 3:

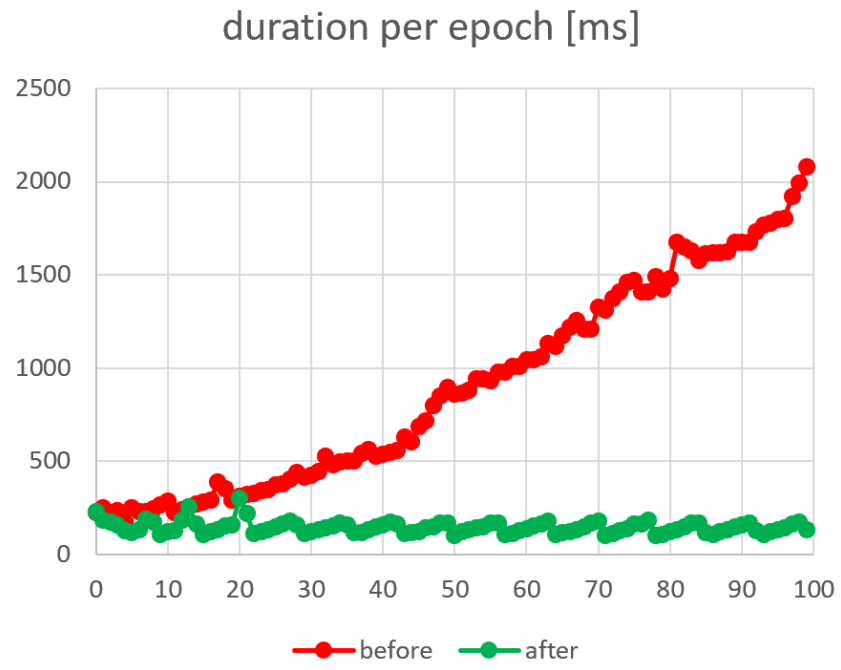
- Improvement: independent global switch to switch garbage collection on.
- More understanding: added a slide (last one) about why there still is a memory jump in the example

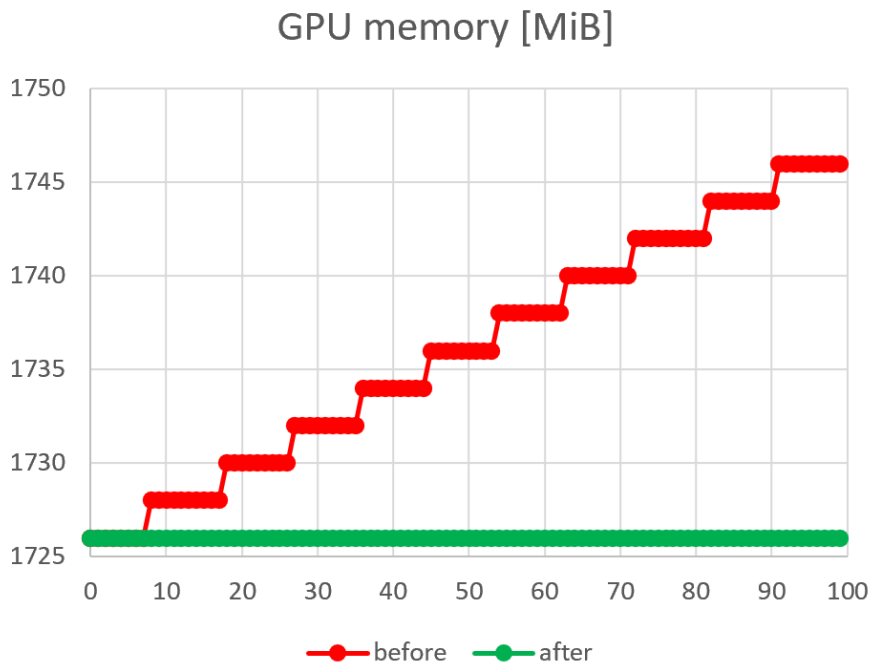
Version 4:

- corrected text on sequence diagram slide 10

Why the memory jump (after) in slide 3

Hypothesis: The frequency of the young generation garbage collections lead to this one time jump
Test: Start the Java JVM with a higher frequency
(only do this for the test - in production accept the jump or keep in mind just in case you tune gc)






```
java -XX:+UnlockExperimentalVMOptions -XX:G1MaxNewSizePercent=30  -jar app/build/libs/app-0.0.1-SNAPSHOT.jar gc
```