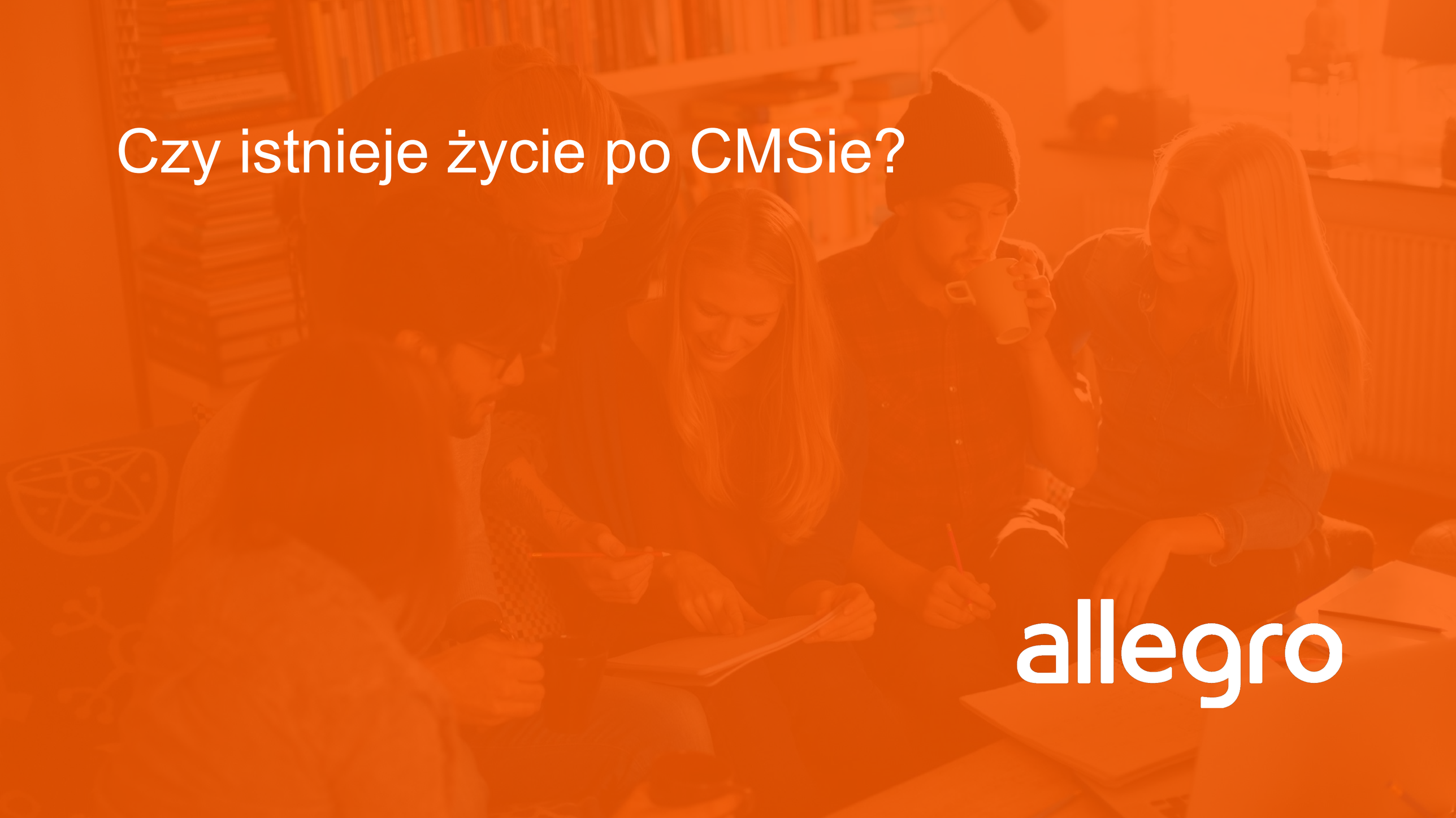
Czy istnieje życie po CMSie?

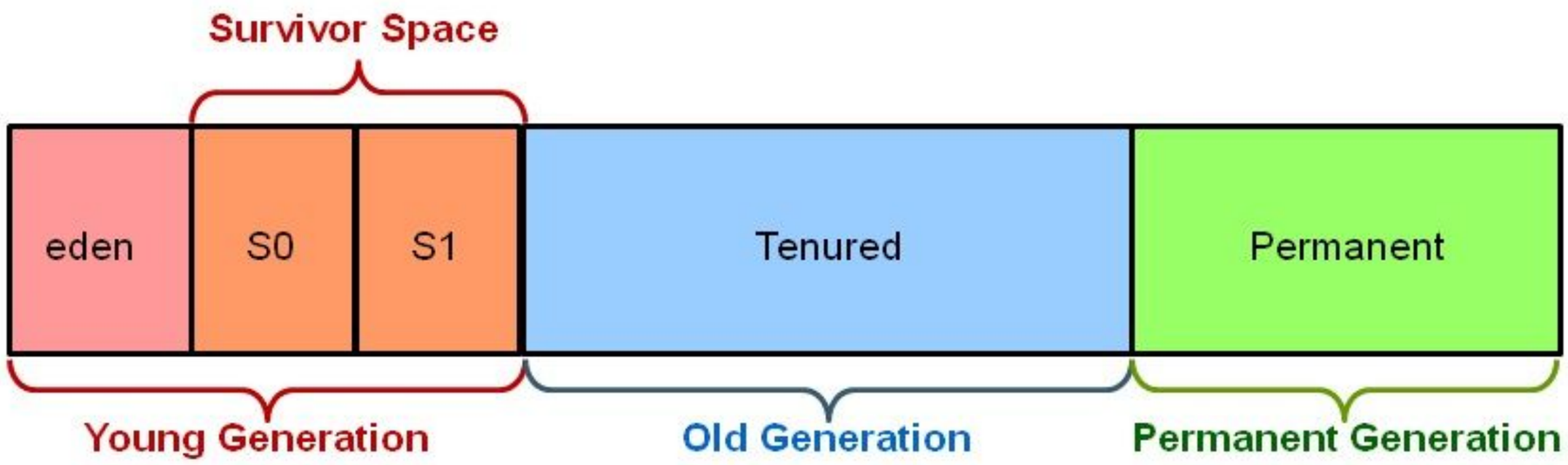


Życie z CMS

- Najbardziej popularny collector dla Javy 1.8 (przynajmniej w Allegro)
- Wystarczający dla potrzeb większości usług
- Część pracy wykonywana w czasie działania aplikacji (concurrent), ale mogą występować długie pauzy, które ciężko przewidzieć

Życie z CMS

- opiera się na modelu pamięci JVM podzielonym na generacje



Java 9

- G1 użyty jako defaultowy collector zamiast ParallelGC
- G1 powstał żeby zastąpić CMS

Czy jesteśmy skazani na G1GC?

Java 11

- wprowadzony Interfejs GC
- stworzenie własnego collectora sprowadza się do napisania implementacji ograniczonej liczby konkretnych interfejsów
- wcześniej trzeba było wiedzieć, które miejsca w kodzie Javy trzeba podmienić

Azul Zing C4

- … albo korzystać z płatnych rozwiązań
- Azul Zing - JVM, dostępna na rynku od ok 2010 roku
- “Ferrari pośród mechanizmów GC”
- używa autorskiego collectora C4 (Continuously Concurrent Compacting Collector)

Azul Zing C4

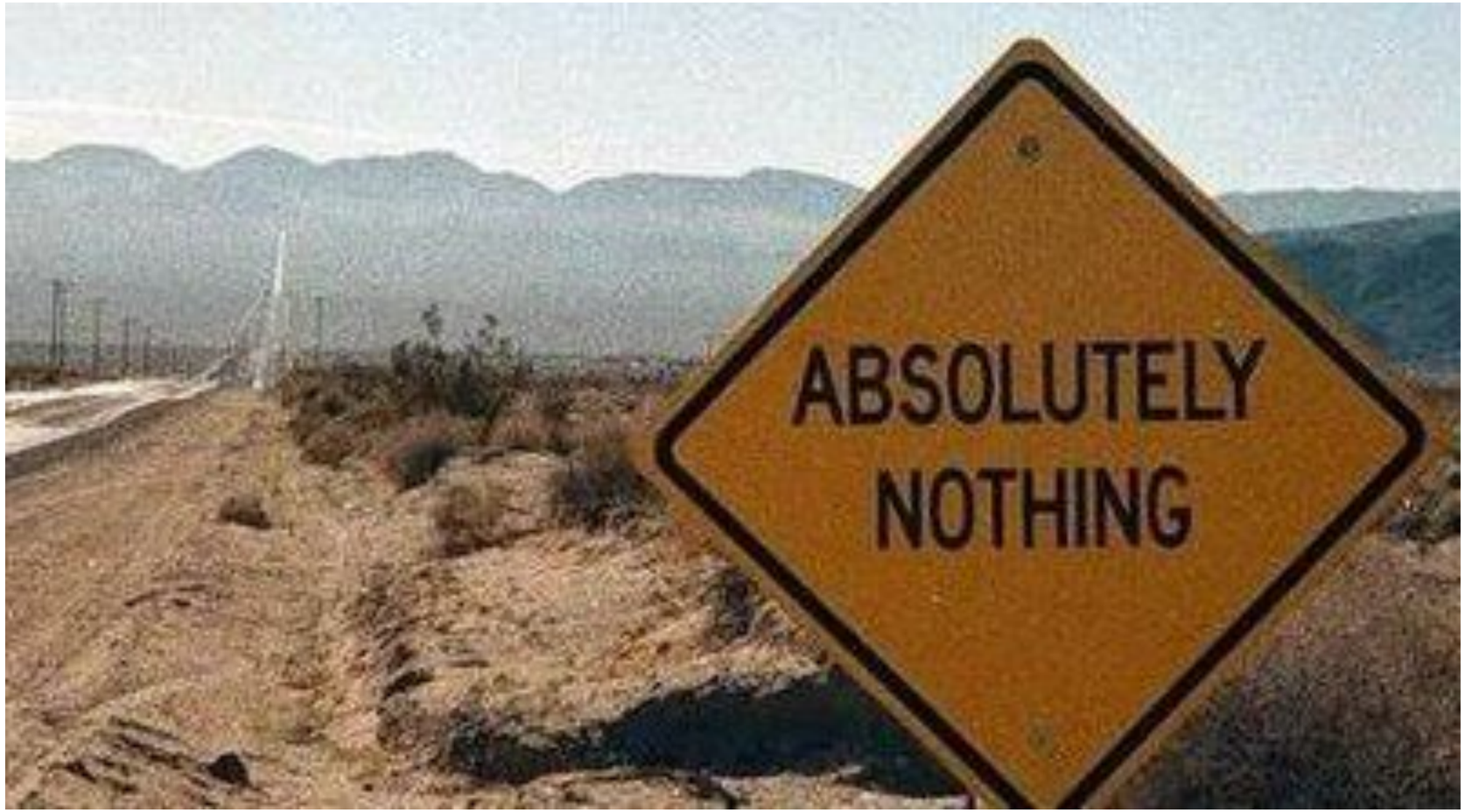
- operuje na takim samym modelu pamięci Heapa jak CMS
- w pełni równoległy (concurrent), nigdy nie powoduje Stop-The-World
- używa mechanizmu barrier do monitorowania obiektów w pamięci dzięki czemu czyszczenie może się odbywać bez pauz


Ale kto by chciał za to płacić?

Epsilon GC

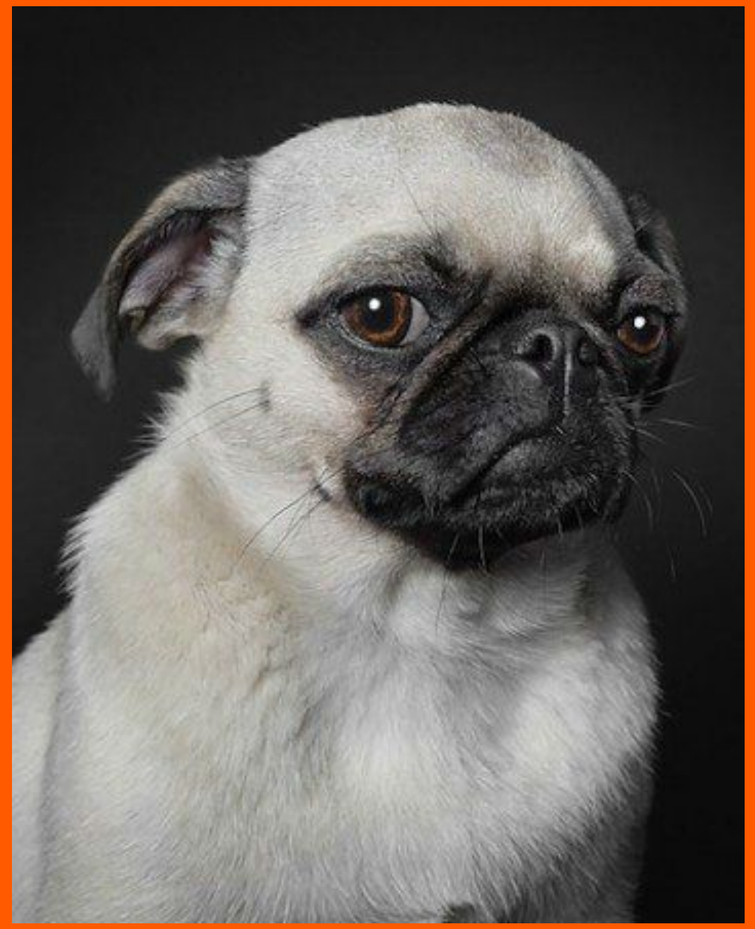
- na szczęście nie trzeba!
- Epsilon GC - dostępny od Javy 11
- Implementacja dzięki interfejsom GC
- Wyjątkowy collector, który robi:

Epsilon GC



Epsilon GC

- no-op garbage collector
- ale może być przydatny:
  - testy wydajności innych mechanizmów GC
  - bardzo krótko żyjące joby
  - testy interfejsu VM
  - testy obciążeń pamięci



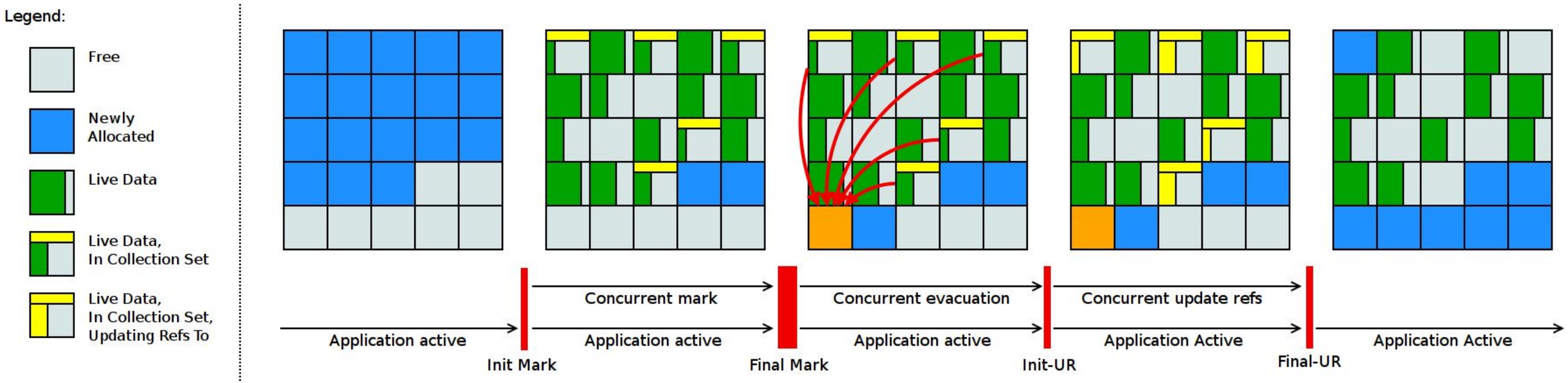
Ale bądźmy poważni


Shenandoah

Shenandoah

"Shenandoah is an ultra-low pause time garbage collector that reduces GC pause times by performing more garbage collection work concurrently with the running Java program. CMS and G1 both perform concurrent marking of live objects. Shenandoah adds concurrent compaction."

Shenandoah



Źródło: https://wiki.openjdk.java.net/display/shenandoah/Main

Shenandoah

- heap podzielony na stałej wielkości regiony zamiast na generacje
- równoległe z działaniem aplikacji etapy oznaczania (marking) i zagęszczania (compaction)
- krótkie pauzy występujące tylko: na początku i końcu oznaczania oraz na początku i końcu aktualizacji referencji
- Collector wciąż w aktywnym rozwoju, od Javy 12 dostępny w ramach OpenJDK


ZGC

ZGC

"The goal of this project is to create a scalable low latency garbage collector capable of handling heaps ranging from a few gigabytes to multi terabytes in size, with GC pause times not exceeding 10ms."

ZGC

- podobny do Shenandoah, też oparty na regionach
- główna różnica: regiony mogą mieć różną wielkość
- krótkie pauzy potrzebne tylko: na początku oznaczania, na koniec oznaczania i na początku relokacji
- jedno z głównych założeń: pauzy zawsze są krótsze niż 10 ms niezależnie od wielkości heapa (nawet dla heapów mierzących kilka TB)
- także aktywnie rozwijany, dostępny w OpenJDK od Javy 11

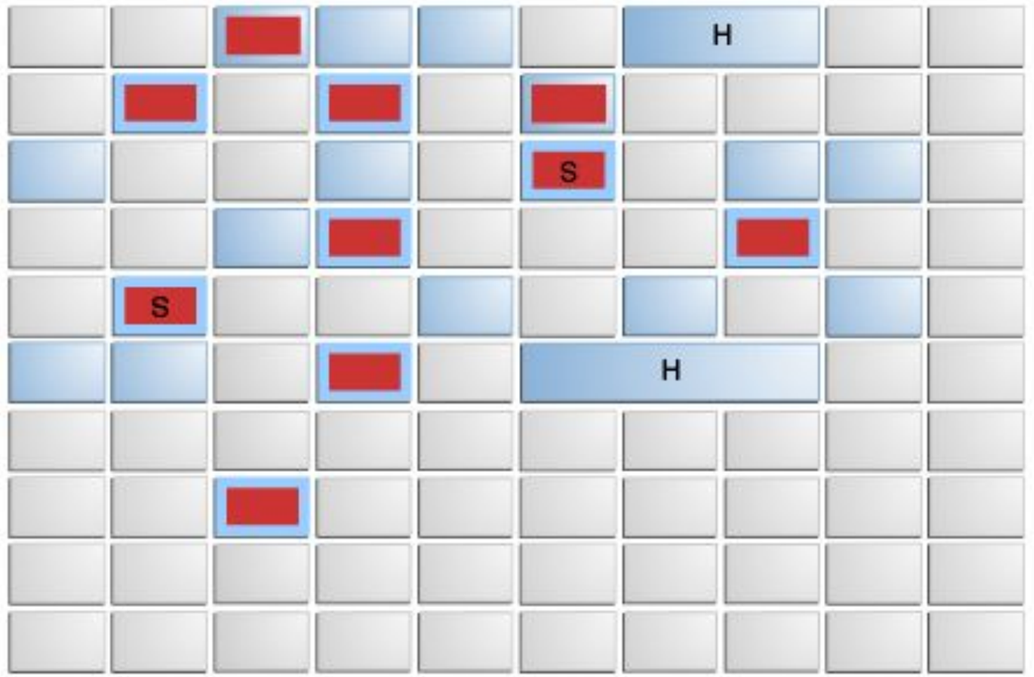

Ale to dopiero przyszłość
(choć niedaleka)

G1GC

"The Garbage-First (G1) garbage collector is targeted for multiprocessor machines with a large amount of memory. It attempts to meet garbage collection pause-time goals with high probability while achieving high throughput with little need for configuration. G1 aims to provide the best balance between latency and throughput using current target applications and environments."

G1GC

- dzieli heap na regiony, ale przydziela im “klasyczne” role:
  - eden
  - survivor
  - old
- łatwiej zarządzać wielkością generacji
- obiekty większe niż 50% regionu trafiają od razu do olda



G1GC

- czyszczenie młodych generacji powoduje pauzy
- gdy zostanie przekroczona pewna granica zajętości heap wykonywany jest “mixed gc” - jednocześnie na regionach young i old, współdzieląc pauzy
- obiekty, które przeżyją z danego regionu są przenoszone do innego, dzięki czemu zagęszczanie odbywa się ciągle
- ale w razie konieczności może być wykonany Full GC, który zatrzymuje aplikację (od Javy 10 jest wielowątkowy)

G1GC

- algorytm próbuje pilnować, żeby pauzy nie były większe niż zadane maksimum
- na podstawie danych z działania aplikacji określa ile regionów na raz może czyścić żeby nie przekroczyć czasu określonego na pauzę
- collector ani nie ma dawać najniższego latency ani najwyższego throughput, ale być gdzieś po środku dla typowych aplikacji, bez konieczności wgryzania się w konfigurację

G1GC

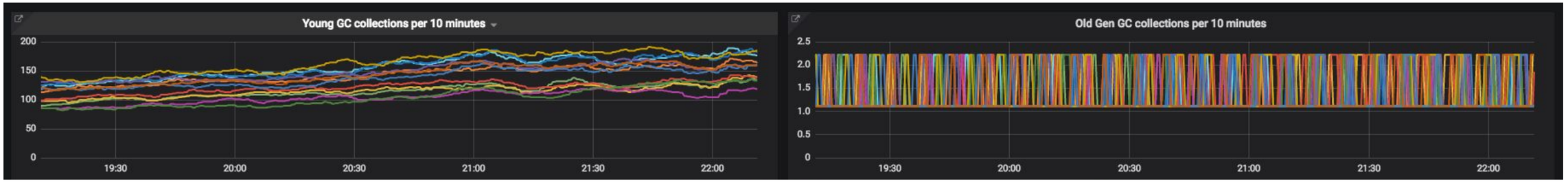
- przejście na G1 jest proste: wyrzuć wszystkie ustawienia dotyczące GC oprócz -Xms, -Xmx, dodaj ewentualnie cel dla pauzy (domyślnie 200 ms)
- **nie ustawiaj** wielkości młodej generacji - G1 poradzi sobie dużo lepiej niż Ty ;)
- G1GC zużywa trochę więcej pamięci niż CMS - to też trzeba wziąć pod uwagę

G1GC w Allegro

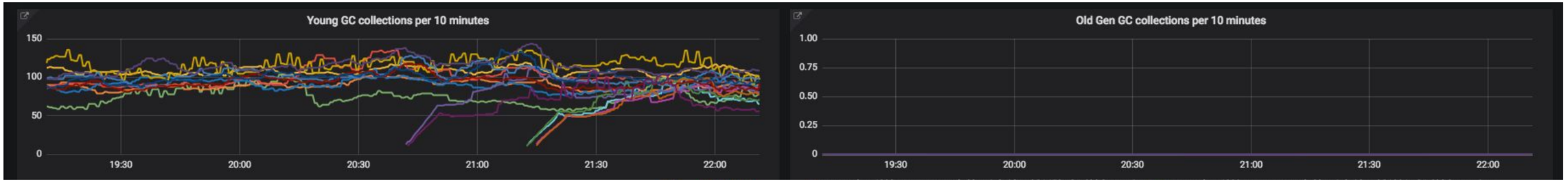
- używamy na Java 10
- wzrost meta-space ok 100 MB
- wpływ na GC:

Liczba cyklów GC

CMS

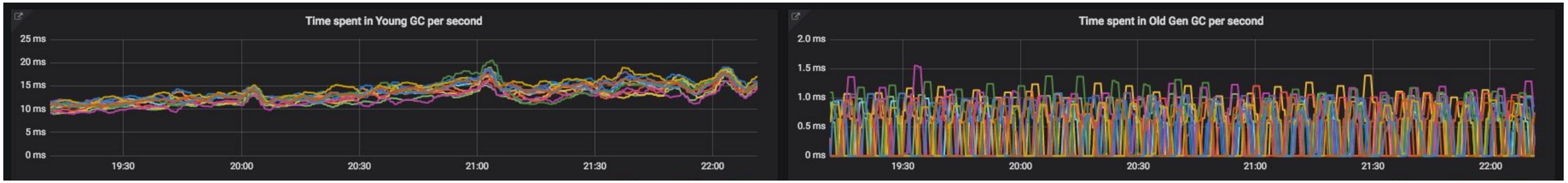


G1

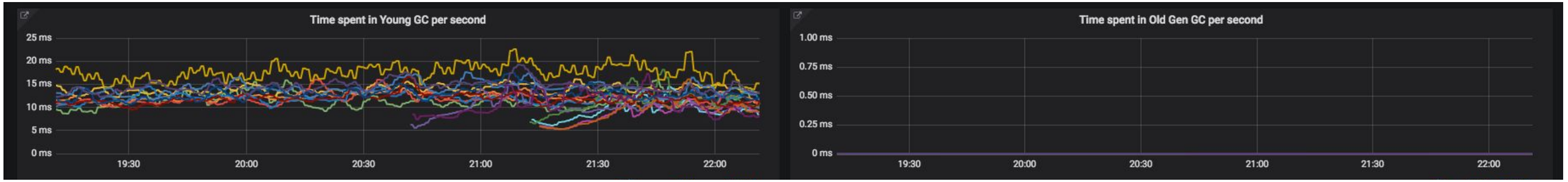


Czas spędzony na GC

CMS



G1



Podsumowując

- G1GC wygląda bardzo dobrze jako następca CMS (od Javy 10)
- warto się przyglądać rozwojowi ZGC i Shenandoah

A idzie jeszcze nowsze...

Pytania?

Dzięki!

allegro Tech

allegro Tech - https://allegro.tech/blog/

 - https://www.meetup.com/allegrotech/

 - https://www.facebook.com/allegro.tech

 - @AllegroTech