Память в Java.
Garbage collector

для JuJa

автор: Александра Дмитренко

Поговорим о том,

1. Сколько места и на что простые типы и объекты тратят, и в чем сложность их хранения.
2. Как в памяти JVM хранится программа.
3. Как работает сборщик мусора.
4. Увидим VirtualVM, как смотреть на занятость памяти.

Модель памяти языка программирования

Это баланс между удобностью использования языка программирования (ЯП) и железяками, на которых ЯП должен работать.

План размышлений

1. Что хотим положить (ведь Java полна разных сущностей)
2. Куда надо класть (структура памяти в разных процессорах и операционках разная)

Как простые типы в Java ложатся на процессор

Type	Contains	Default	Size	Range
`boolean`	`true` or `false`	`false`	1 bit	NA
`char`	Unicode character	`\u0000`	16 bits	`\u0000` to `\uFFFF`
`byte`	Signed integer	0	8 bits	-128 to 127
`short`	Signed integer	0	16 bits	-32768 to 32767
`int`	Signed integer	0	32 bits	-2147483648 to 2147483647
`long`	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
`float`	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
`double`	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

Пример: int -> 2^{32} / 2 = 2147483648

Как считать единички и нолики в разрядной сетке

Значения в разрядной сетке - метки, необходимо ли считать степень 2. Прочитаем с конца:

$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 1 + 8 + 32 + 64 = 105$

Красные - лишние для подсчета числа, ведь в разрядной сетке стоят нули. Максимально можем закодировать число: $2^7 - 1 = 128 - 1 = 127$

Вопросы

1. Число какого простого типа данных только что было закодированное? (127)
2. Какое максимальное число можно закодировать в 16ти разрядах?

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

3.* Как записать 64 битный long на 32 битной ОС?

Последствие записи сущностей, превышающих разрядную сетку

Нарушение атомарности: запись будет производиться в 2 захода.

Чем чревато: При многопоточности в момент записи значение может быть как не старым, так и не новым.

Решение: volatile long, volatile double - обеспечивает, чтоб другой поток не читал изменения, пока первый их не закончит.

Минус: платим производительностью. Атомарное чтение, а тем более synchronized блок будут работать медленнее.

Вопрос: надо ли писать volatile long в программе для 64 битого процессора?

Почему Java не полностью ООП язык?

В Java все объекты **кроме** примитивов и ссылок на объекты

Сколько места занимают объекты в Java

Объект состоит из

- заголовка (2 маш. слова: 8 байт для 32х и 16 для 64х разрядной сетки + 4 байта на длину массива)
- простых типов
- ссылочных типов - объектов
- смещения для выравнивания к машинному слову

Размер строки

Пусть класс строки содержит следующие поля:

```
    private final char value[];
    private final int offset;
    private final int count;
    private int hash;
```

Тогда ее размер для 32х процессора считается так:

```
new String()
Заголовок: 8 байт
Поля int: 4 байта * 3 == 12 байт
Ссылочная переменная на объект массива: 4 байта
Итого: 24 байта

new char[1]
Заголовок: 8 байт + 4 байта на длину массива == 12 байт
Примитивы char: 2 байта * 1 == 2 байта
Выравнивание для кратности 8 : 2 байта
Итого: 16 байта

Итого, new String("a") == 40 байт
```

Вопрос

Сколько места будет занимать строка из 2х символов?

```
new String("ab")
```

Ответ: 40 байт, столько же, сколько и от одного.

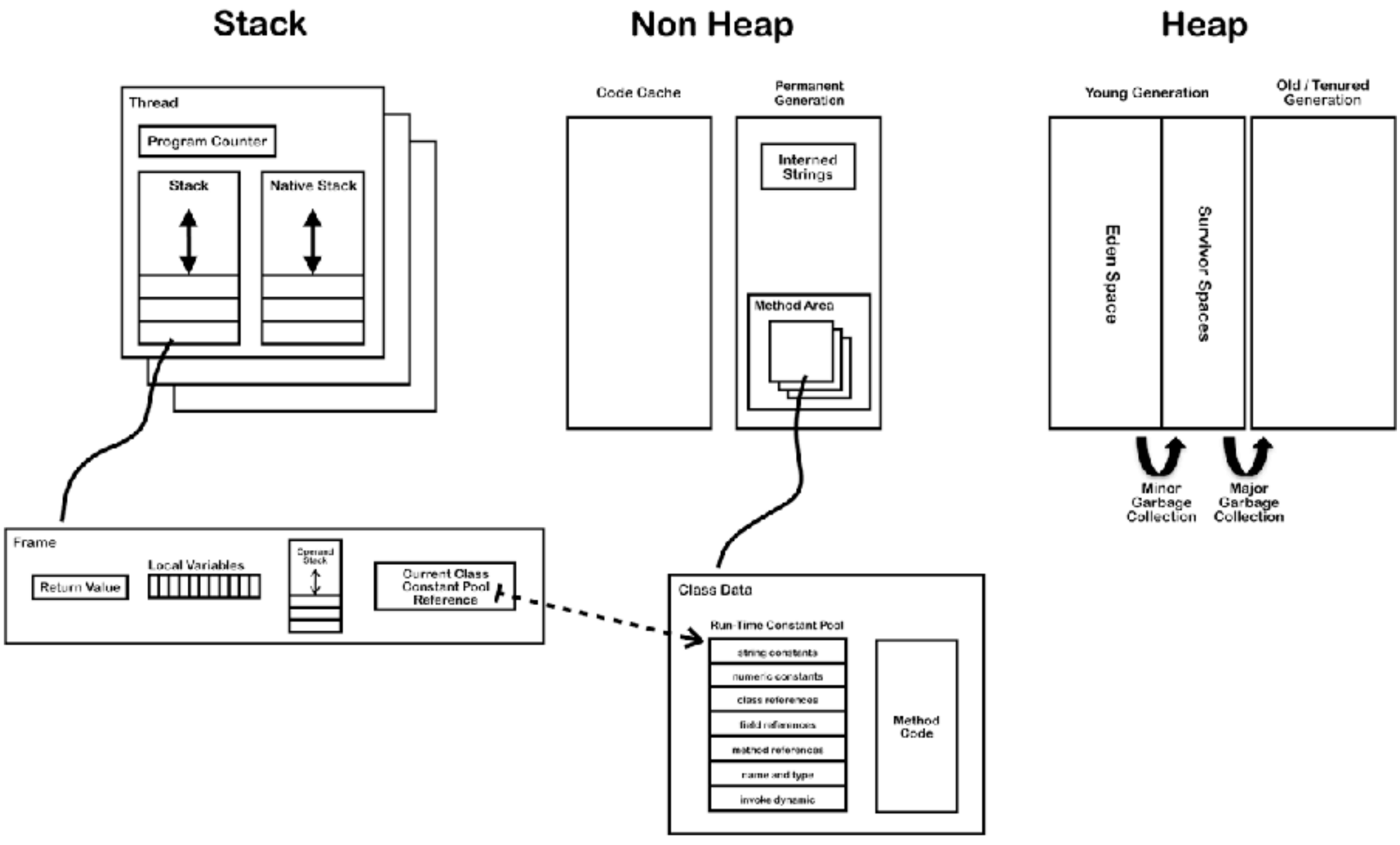
Объяснение: один символ вместится в память, потраченную на выравнивание к длине маш. слова.

Ситуация в C++

short	int	long	ptr	long long	Label		Examples
...	16	...	16	...	IP16		PDP-11 Unix (1973)
16	16	32	16	...	IP16L32		PDP-11 Unix (1977); multiple instructions for long
16	16	32	32	...	I16LP32		MC68000 (1982); Apple Macintosh 68K; Microsoft operating systems (plus extras for x86 segments)
16	32	32	32	...	ILP32		IBM 370; VAX Unix; many workstations
16	32	32	32	64	ILP32LL or ILP32LL64		**Microsoft Win32;** Amdahl; Convex; 1990 Unix systems; Like IP16L32, for same reason; multiple instructions for long long
16	32	32	64	64	LLP64 or IL32LLP64 or P64	64-bit systems	**Microsoft Win64** (X64 / IA64)
16	32	64	64	64	LP64 or I32LP64		**Most Unix systems** (Linux, Solaris, DEC OSF/1 Alpha, SGI Irix, HP UX 11)
16	64	64	64	64	ILP64		HAL; logical analog of ILP32
64	64	64	64	64	SILP64		UNICOS

Для совместимости 64-битная система Windows поддерживает исполнение 32-битных программ, которые работают в режиме модели данных ILP32LL. В коде программ **необходимо учитывать разрядность** используемых данных.

Структура памяти



Каждому потоку по…
Или стек и компоненты

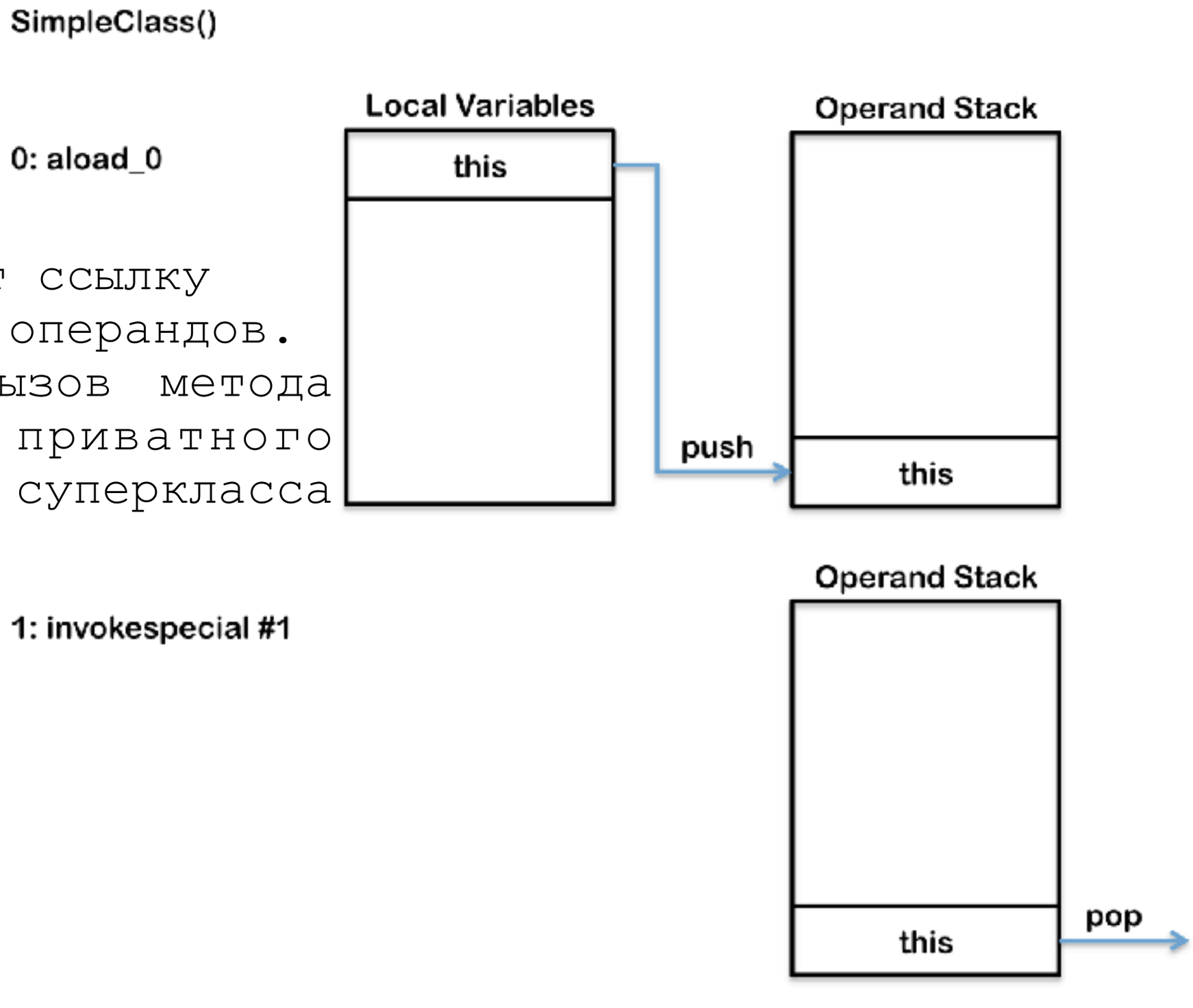
1. Стек с его настройками и последовательностью действий (LIFO)
2. Простые типы
3. Ссылки на объекты

Действия класса идут в стек

```
public class SimpleClass {
        public void sayHello() {
                System.out.println("Hello");
        }
}
```

Работа стека с классом

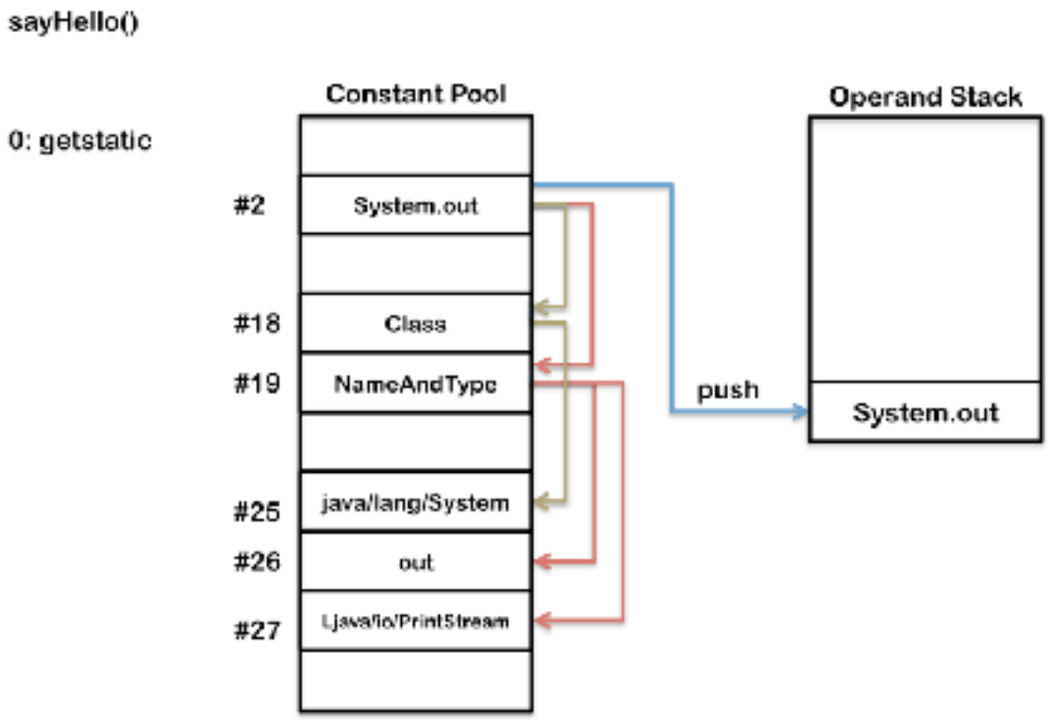
Команды стека:
aoland: Загружает ссылку на объект в стек операндов.
invokespecial: вызов метода инициализации, приватного метода и метода суперкласса объекта.

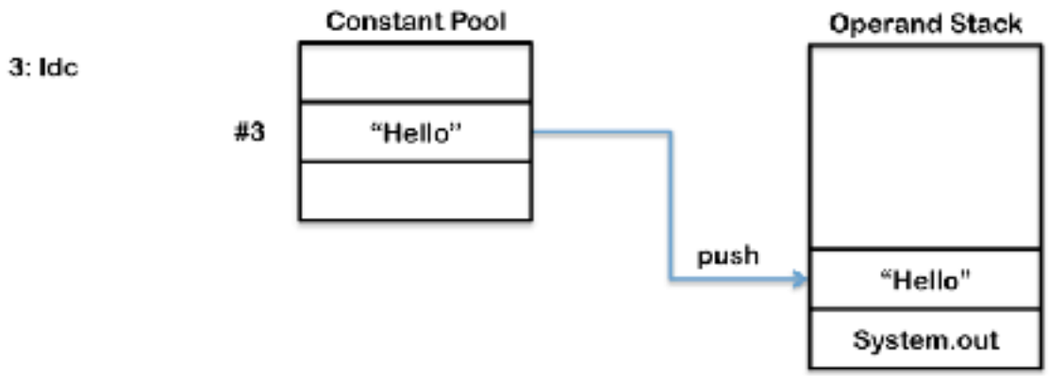


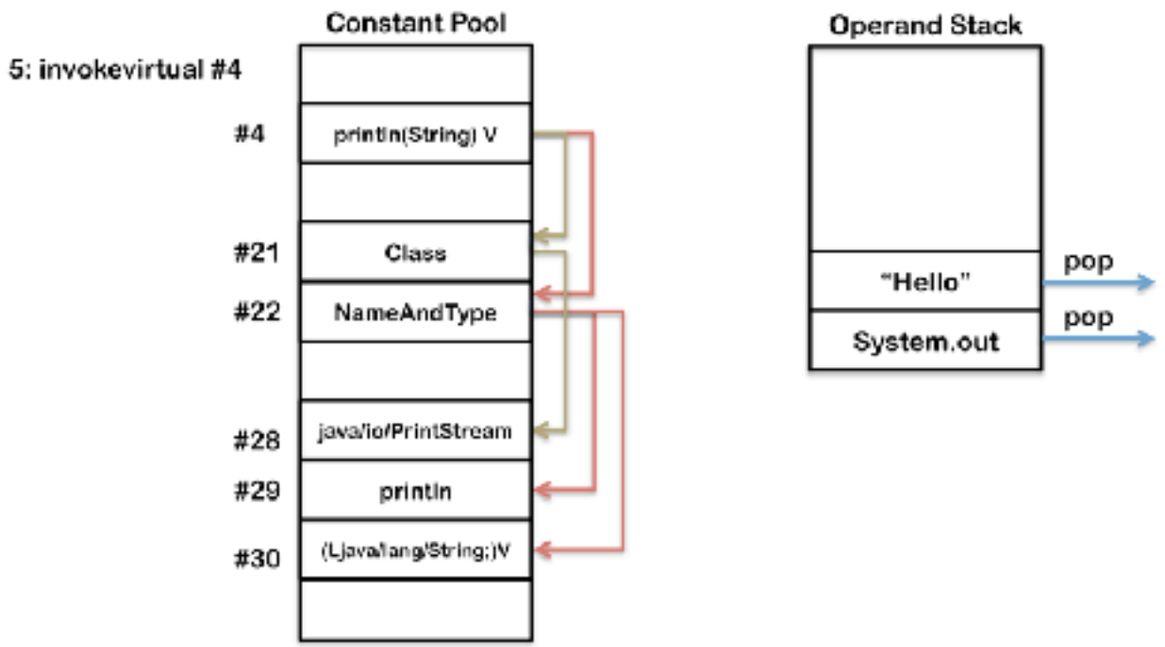
getstatic - запушить статическую переменную из run time constant pool в operand stack.

ldc - запушить константу в operand stack.

invokevirtual - вызов метода, базирующегося на классе объекта







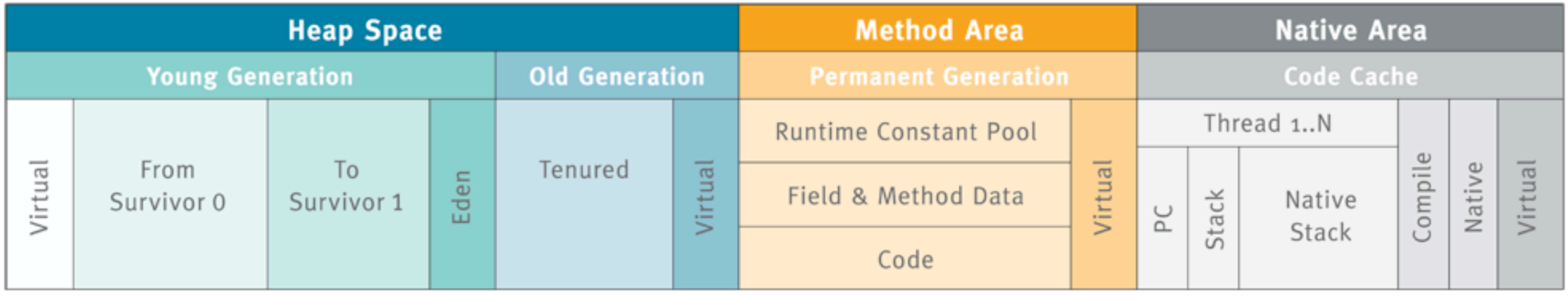
Потоки делят

1. Heap - хранит все объекты и массивы: Young (Eden, Survivor spaces), Old и Tenured Space

2. Non-Heap: Permanent Generations(используемые классы, методы), Method Area, String Pool (interned strings), Code Cache (native code compiled methods by JIT - Just in Time Compilation)

Method Area:

Classloader Reference, Run Time Constant Pool, Field data, Method Data, Method code (byte code) and its exceptions



Eden Space (heap)

- Память под все создаваемые из программы объекты.
- Живут недолго (итераторы, объекты методов).
- GC выполняет быструю (minor collection) сборку мусора, которая удаляет почти все и часть перемещает в область выживших объектов.

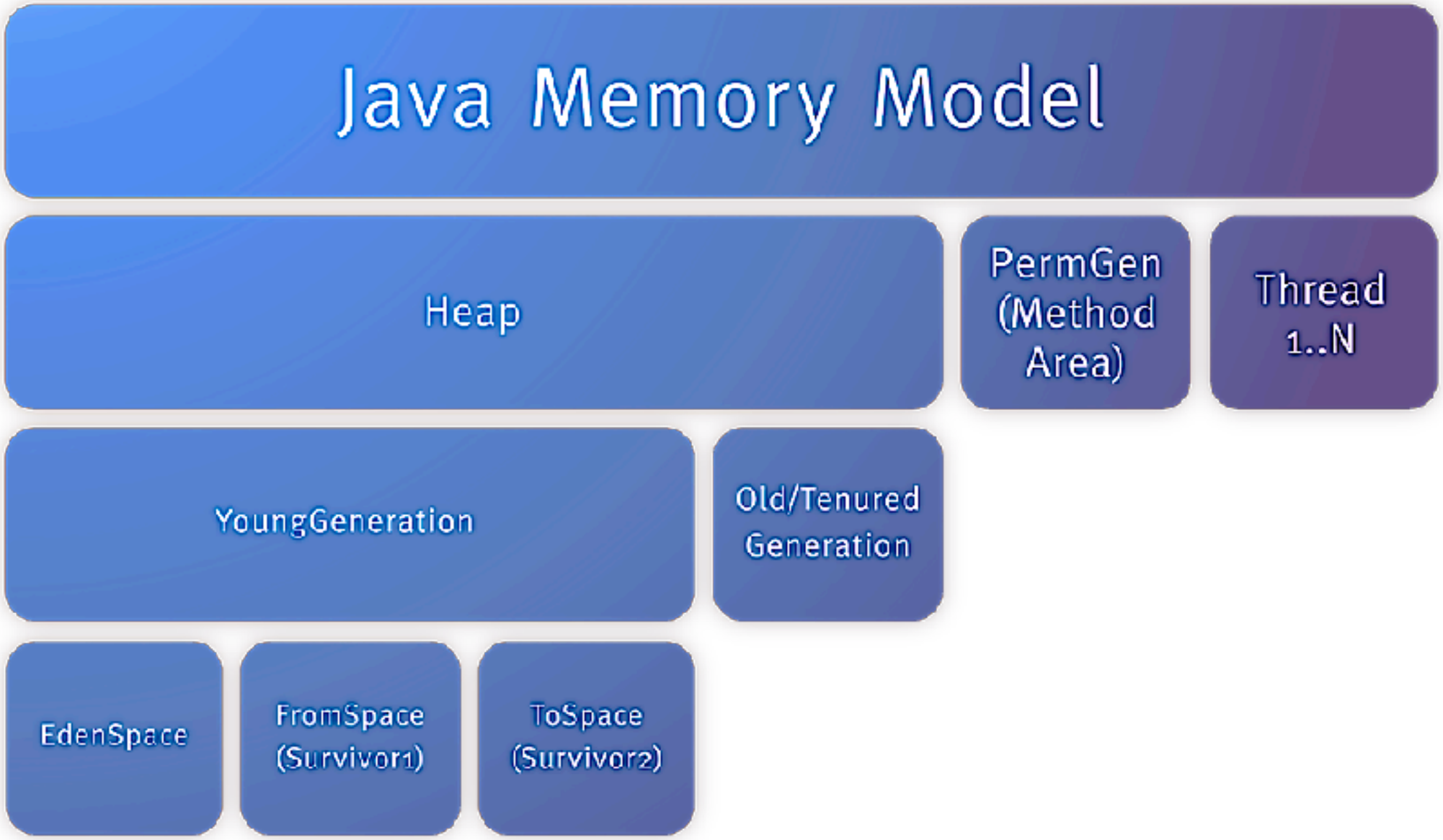
Survivor и Tenured Space (heap)

Survivor Space – содержит объекты, пережившие Eden. Время от времени долгоживущие объекты перемещаются в Tenured Space.

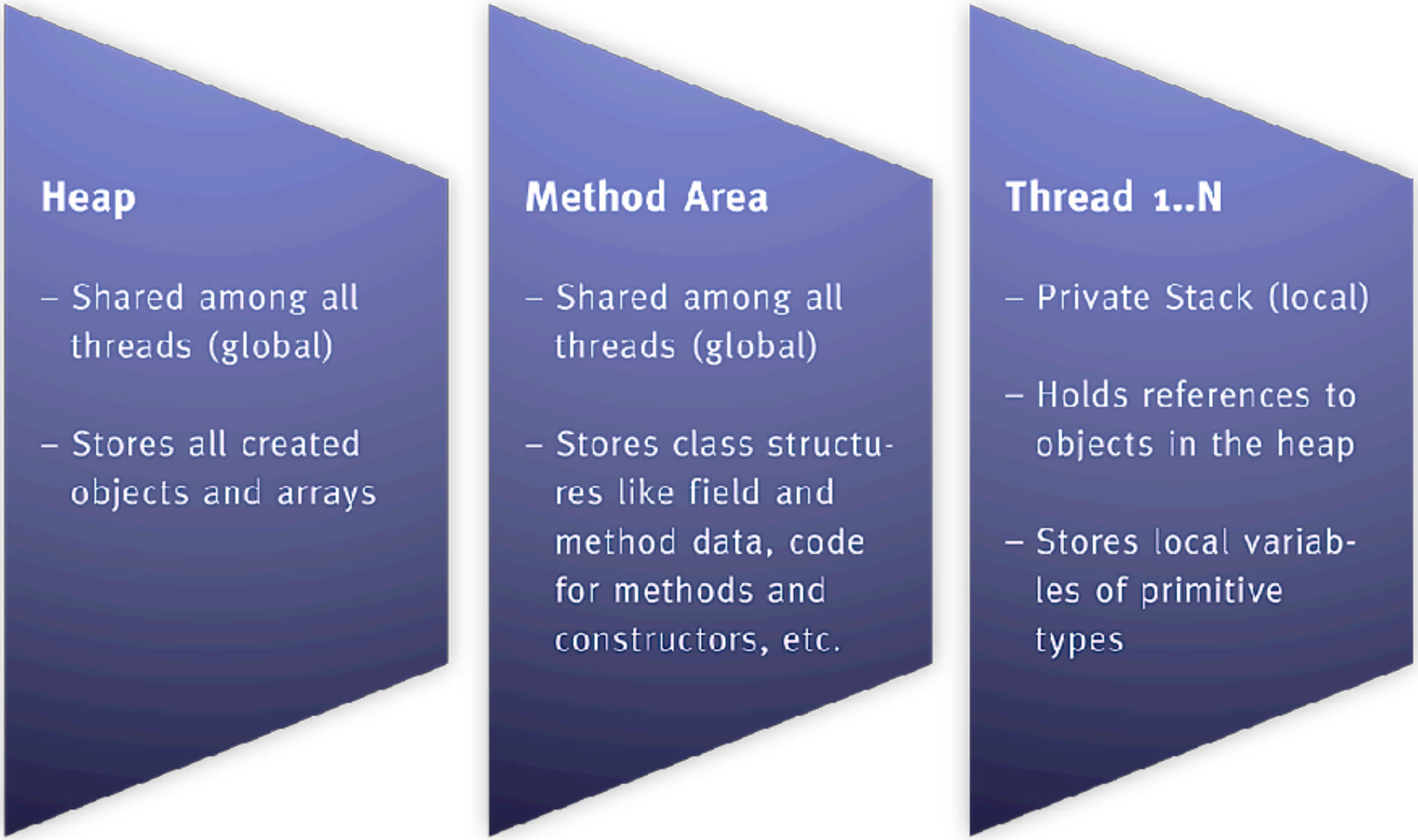
Tenured (Old) Generation (heap) - долгоживущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и проч.).

Когда заполняется эта область, выполняется полная сборка мусора (full, major collection), которая обрабатывает все созданные JVM объекты.

Структура памяти = Heap + Non-Heap



Подсумируем самое главное про память



Разберем, в какую область памяти что когда записывается

```
public static void main(String[] args) { // строка 1
    int i=1; // строка 2
    Object object = new Object(); // строка 3
    Memory memory = new Memory(); // строка 4
    memory.exMethod(object); // строка 5
} // строка 9

private void exMethod(Object param) { // строка 6
    String string = param.toString(); // строка 7
    System.out.println(string );
} // строка 8
```

Последовательность выполнения программы

1. Все необходимые для запуска программы классы загружаются в память.

2. Команды-действия-строки записываются в стек в порядке выполнения.

3. В `строке 2` создается int’овая переменная, которая хранится в памяти стека метода `main()`.

4. В куче создается объект в `строке 3`. Стековая память содержит ссылку на него. Точно такой же процесс происходит, когда мы создаем объект Memory в `строке 4`.

Как выглядит выполнение Java программы

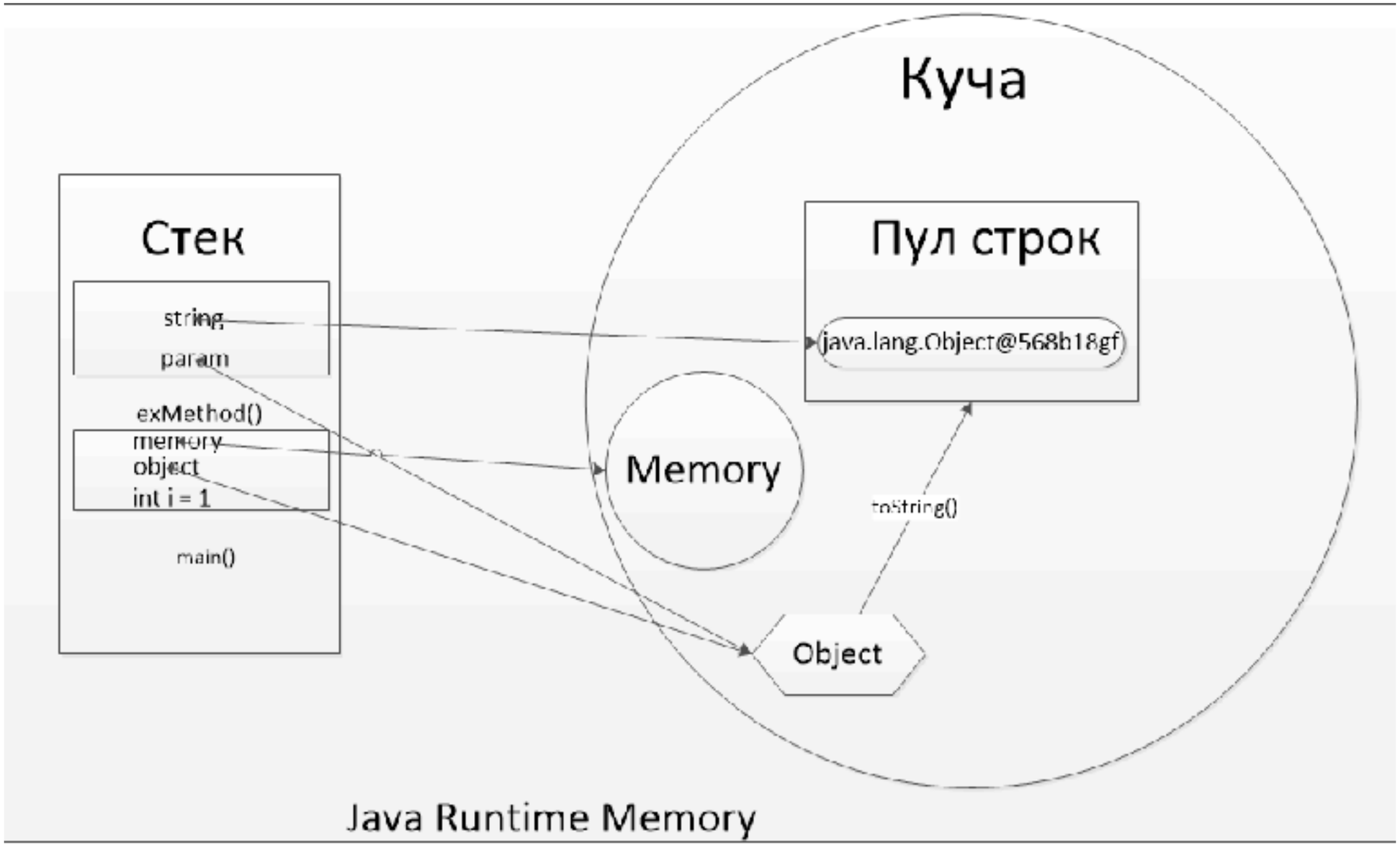
5. В `строке 5` под метод `exMethod()` создается блок (frame) на вершине стека. В Java объекты передаются по значению ссылки, а примитивы по значению, в `строке 6` не будет создана новая ссылка на объект, созданный в строке 3.

6. Строка, созданная в `строке 7`, отправляется в Пул строк (String Pool). На эту строку также создается ссылка в стековой памяти метода `exMethod()`.

7. Метод `exMethod()`завершается на `строке 8`, поэтому блок стековой памяти для этого метода становится свободным.

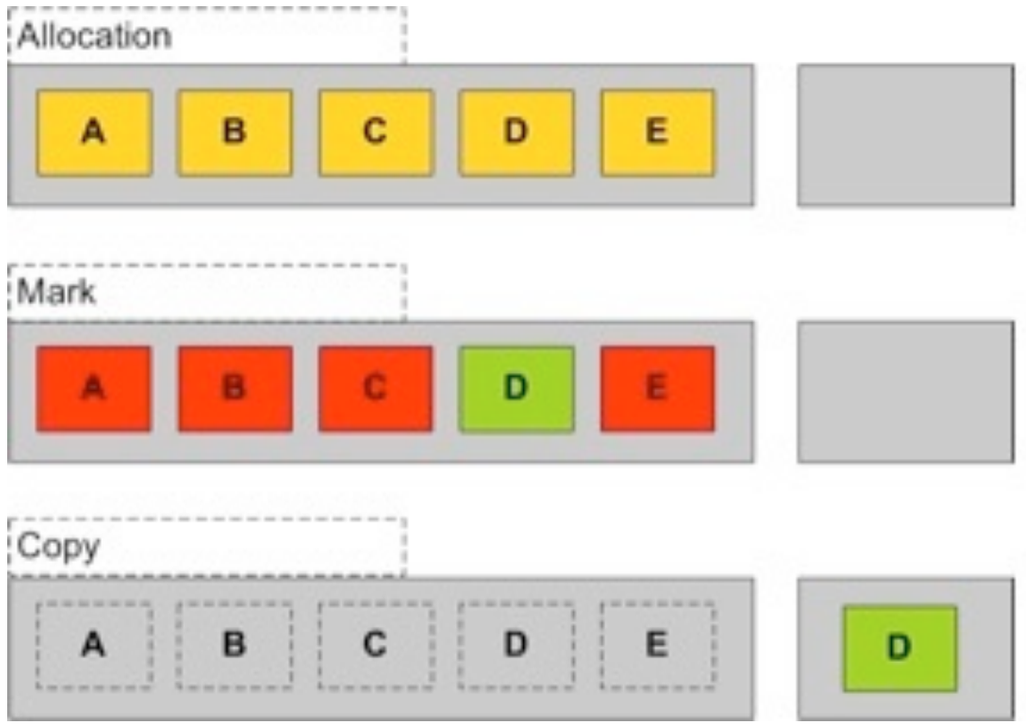
8. В `строке 9` метод `main()` завершается, поэтому стековая память для метода `main()` будет уничтожена. Также программа заканчивается в этой строке, следовательно, Java Runtime освобождает всю память и завершает программу.

Стек хранит последовательность
Куча - объекты



GC. CopyCollection

Просматривает неиспользуемые объекты. Выжившие отправляет в Survivor. Хорош, когда много мусора и мало полезных объектов

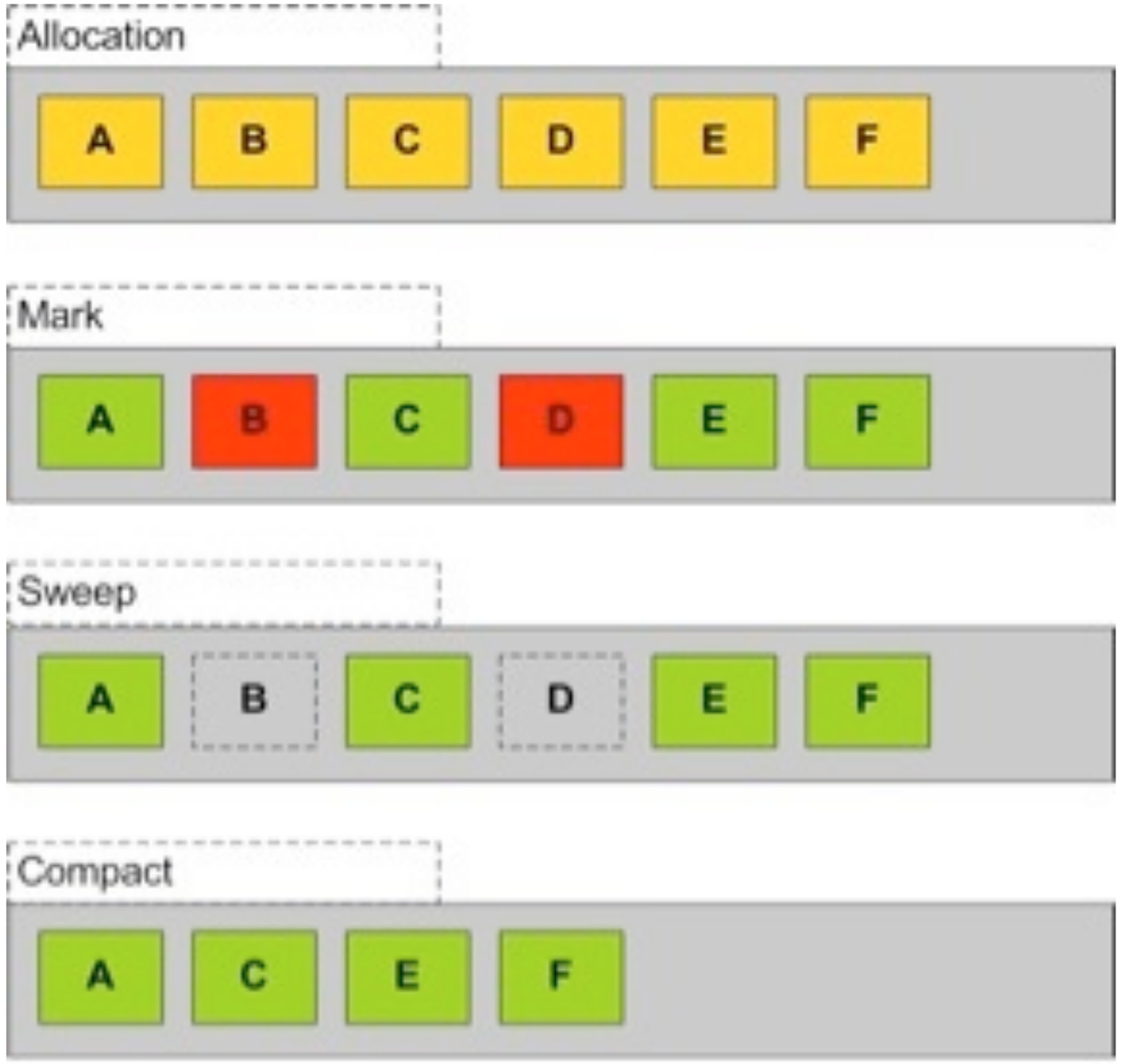


GC.Mark-Sweep-Compact Collection

1) «Mark»: помечаются неиспользуемые объекты (красные).

2) «Sweep»: эти объекты удаляются из памяти. Обратите внимание на пустые слоты на диаграмме.

3) «Compact»: объекты размещаются, занимая свободные слоты, что освобождает пространство на тот случай, если потребуется создать «большой» объект.



О стиле написания кода

- При частом использовании JVM делает Hot Spots для методов в Method Area и не удаляет их, если они уже якобы не нужны. «Часто» начинается с 10 000 вызовов.
  Работает только для методов, у которых менее 35 строк.

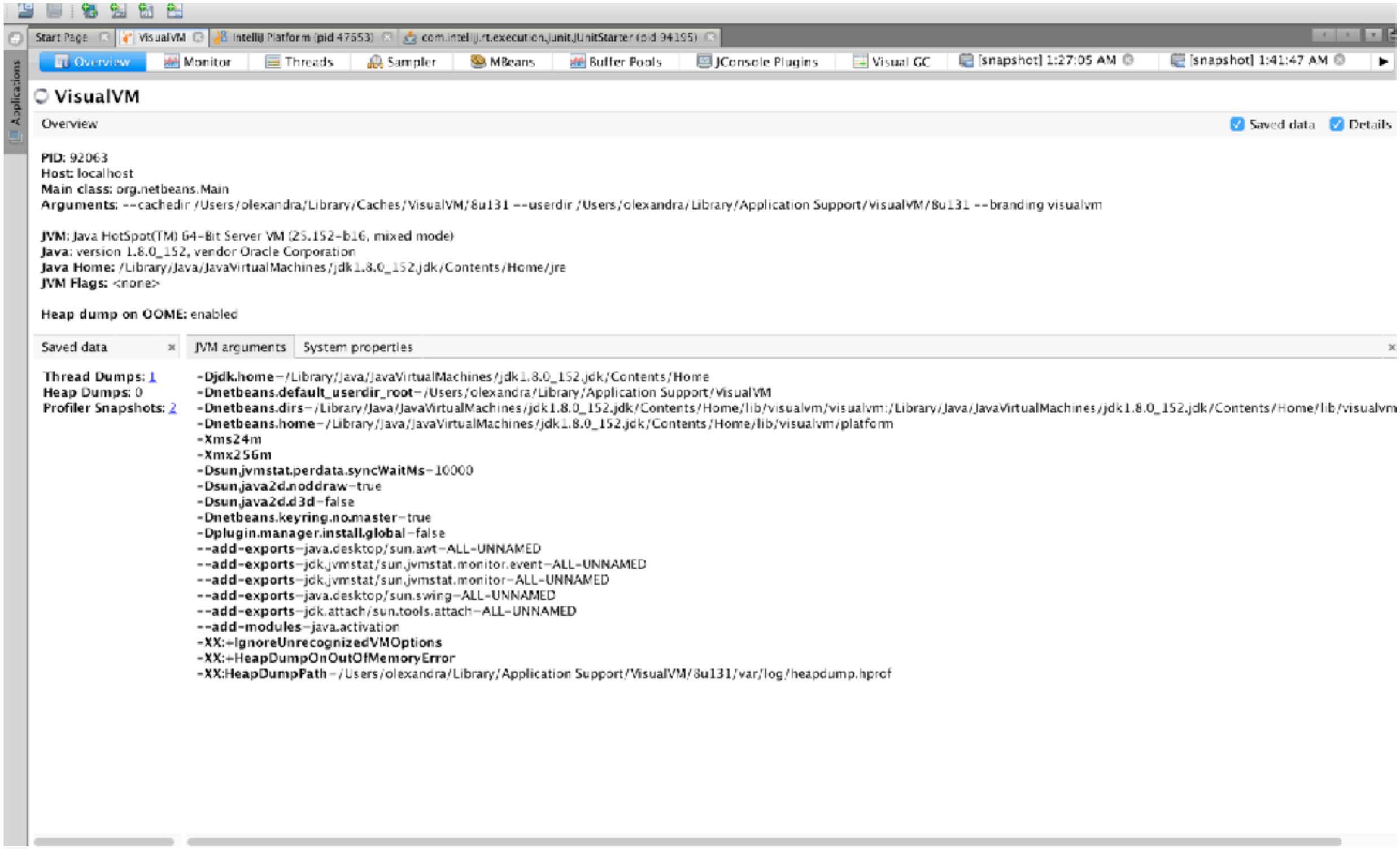
- Constant folding - разворачивание объекта в простые типы: Point<x,y> -> int x, int y. Делается для упрощения вычислений.

Как посмотреть, есть ли утечки в вашей программе

- В Java 6+ есть инструмент VisualVM для визуализации, что происходить с JVM во время ее работы.

- Запустить: в консоли набрать jvisualvm
  Путь к программе: Path_to_Java/bin/jvisualvm

VisualVM.Overview



Маленькая но грозная програмка

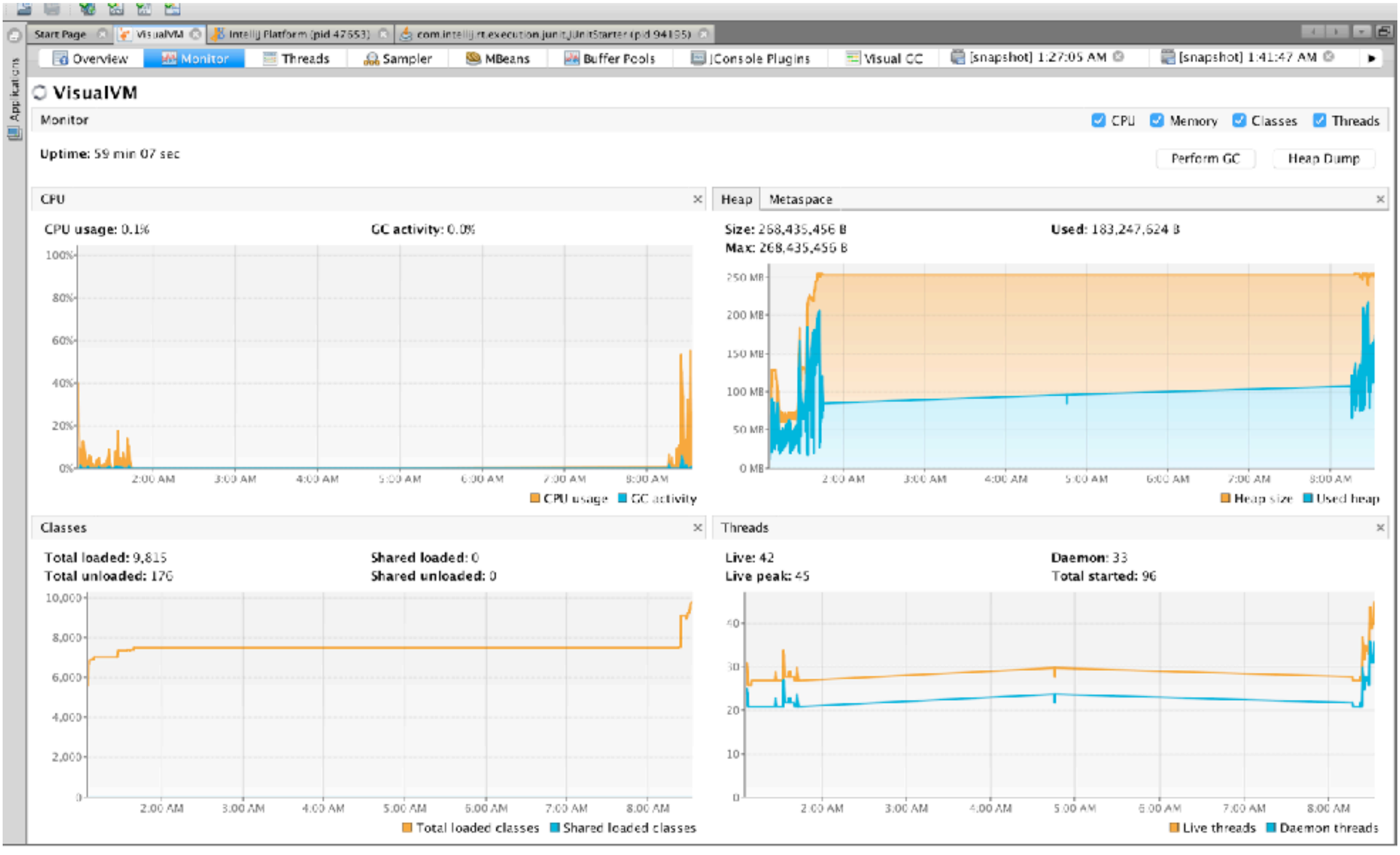
```
@Test
public void eternalCycle() {
    String a = "Very long string";
    int j = 1;
    while (true) {
        int i = j + 1;
        a = a + i;
        j = i;
    }
}
```

Как можно получить OutOfMemoryError?

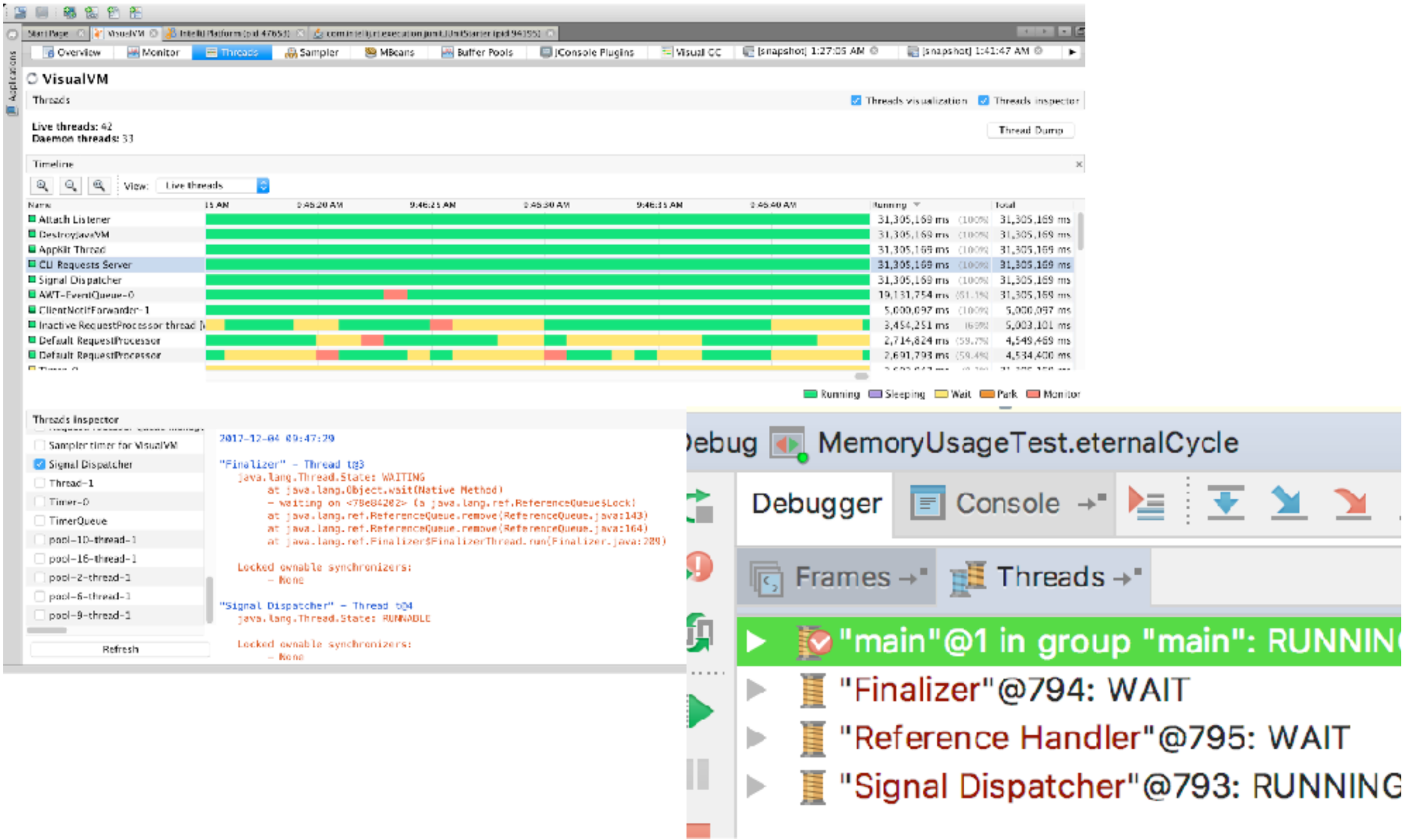
В классе AbstractStringBuilder есть метод, который выделяет память под новый массив символов в String

```
void expandCapacity(int minimumCapacity) {
    int newCapacity = value.length * 2 + 2;
    if (newCapacity - minimumCapacity < 0)
        newCapacity = minimumCapacity;
    if (newCapacity < 0) {
        if (minimumCapacity < 0) // overflow
            throw new OutOfMemoryError();
        newCapacity = Integer.MAX_VALUE;
    }
    value = Arrays.copyOf(value, newCapacity);
}
```

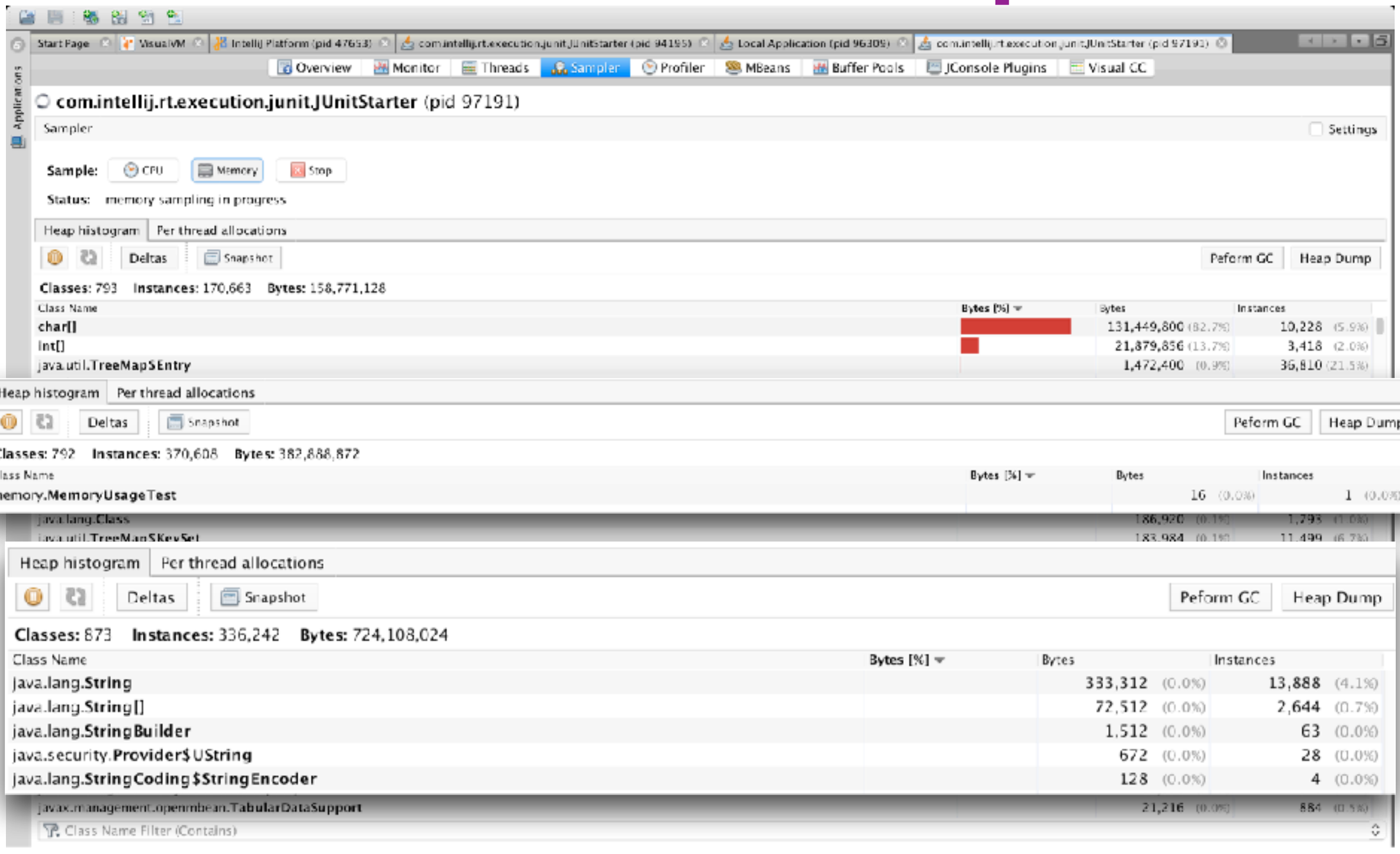
VisualVM.Monitor



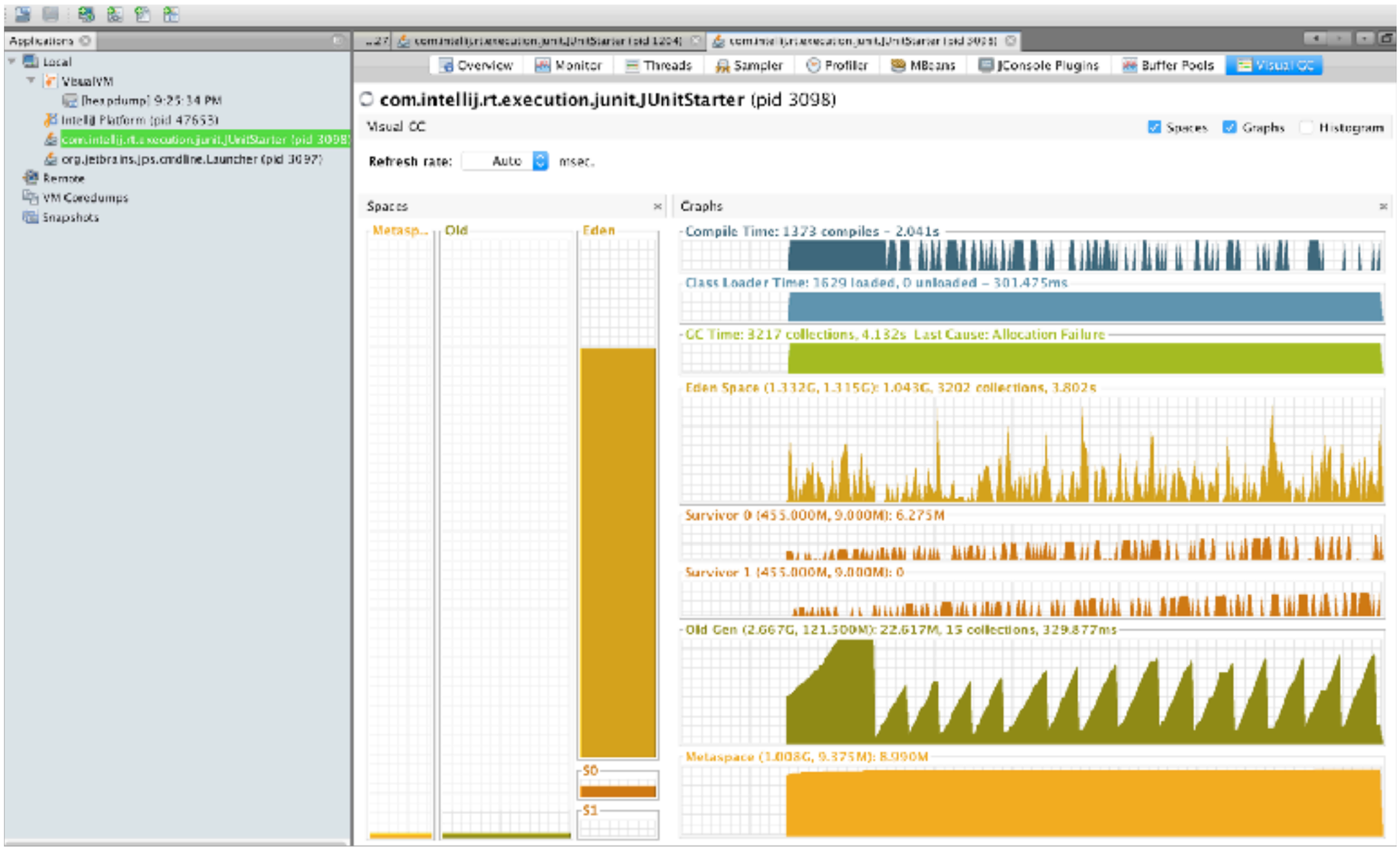
VisualVM.Threads



VisualVM.Heap



GC справляется



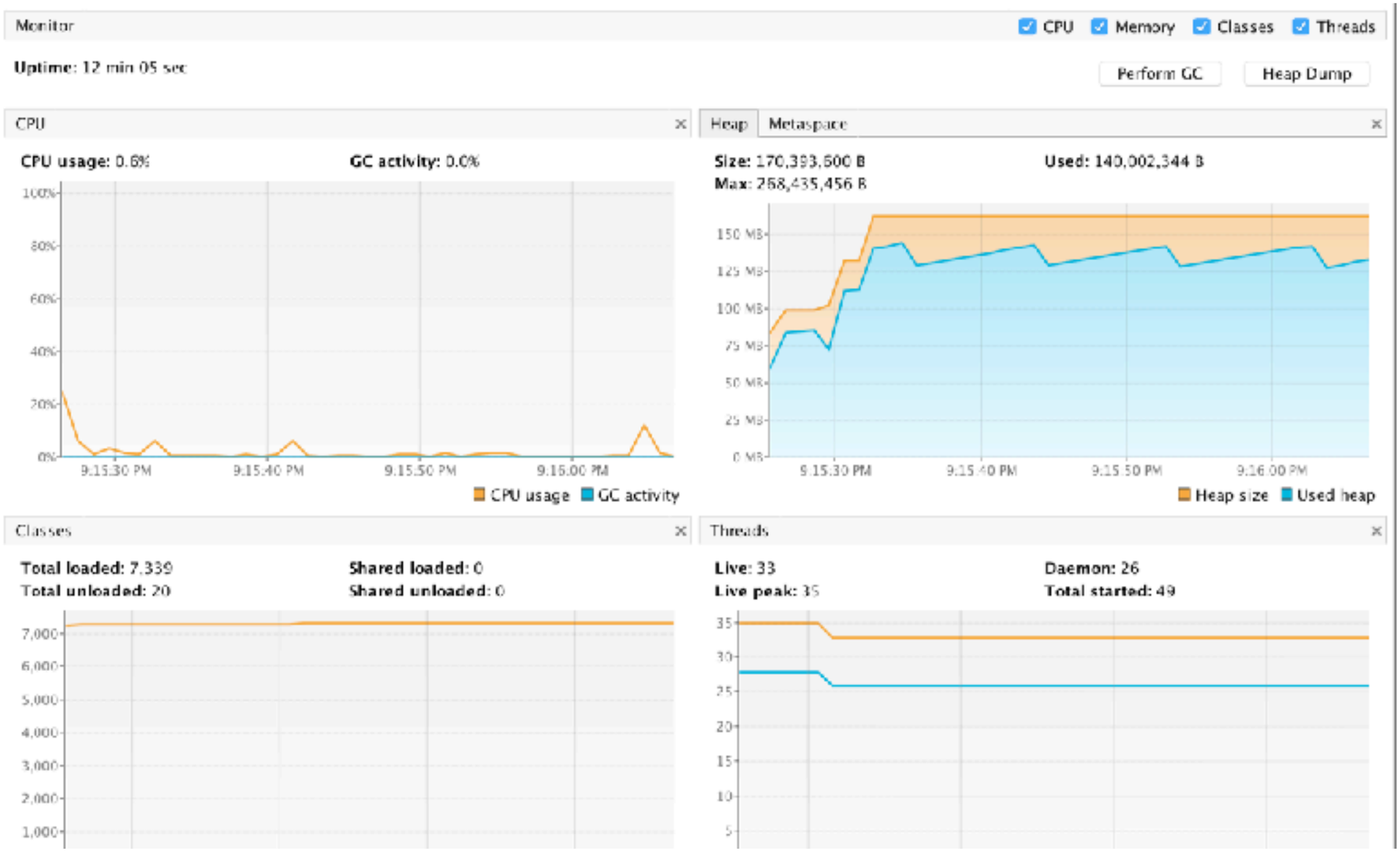
А теперь будем прибавлять строки

```
@Test
public void eternalCycleWithMyString(){
    MyString string = new MyString("My very long string");
    while (true){
        string.setMyString(string.getMyString() + string.getMyString());
    }
}
```

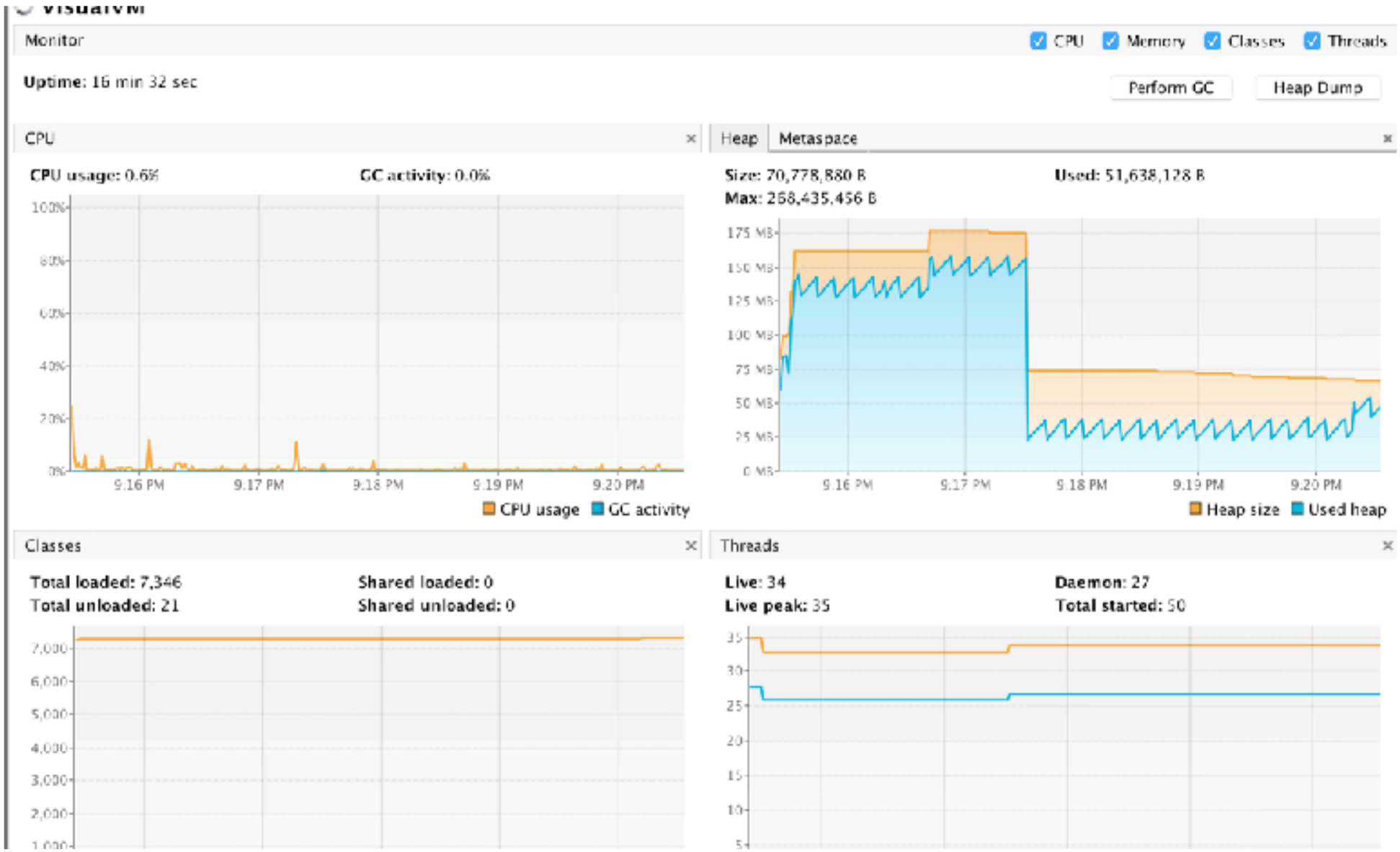
```
public class MyString {
    private String myString;

    public MyString(String myString) {
        this.myString = myString;
    }
    public String getMyString() {
        return myString;
    }
    public void setMyString(String myString) {
        this.myString = myString;
    }
}
```

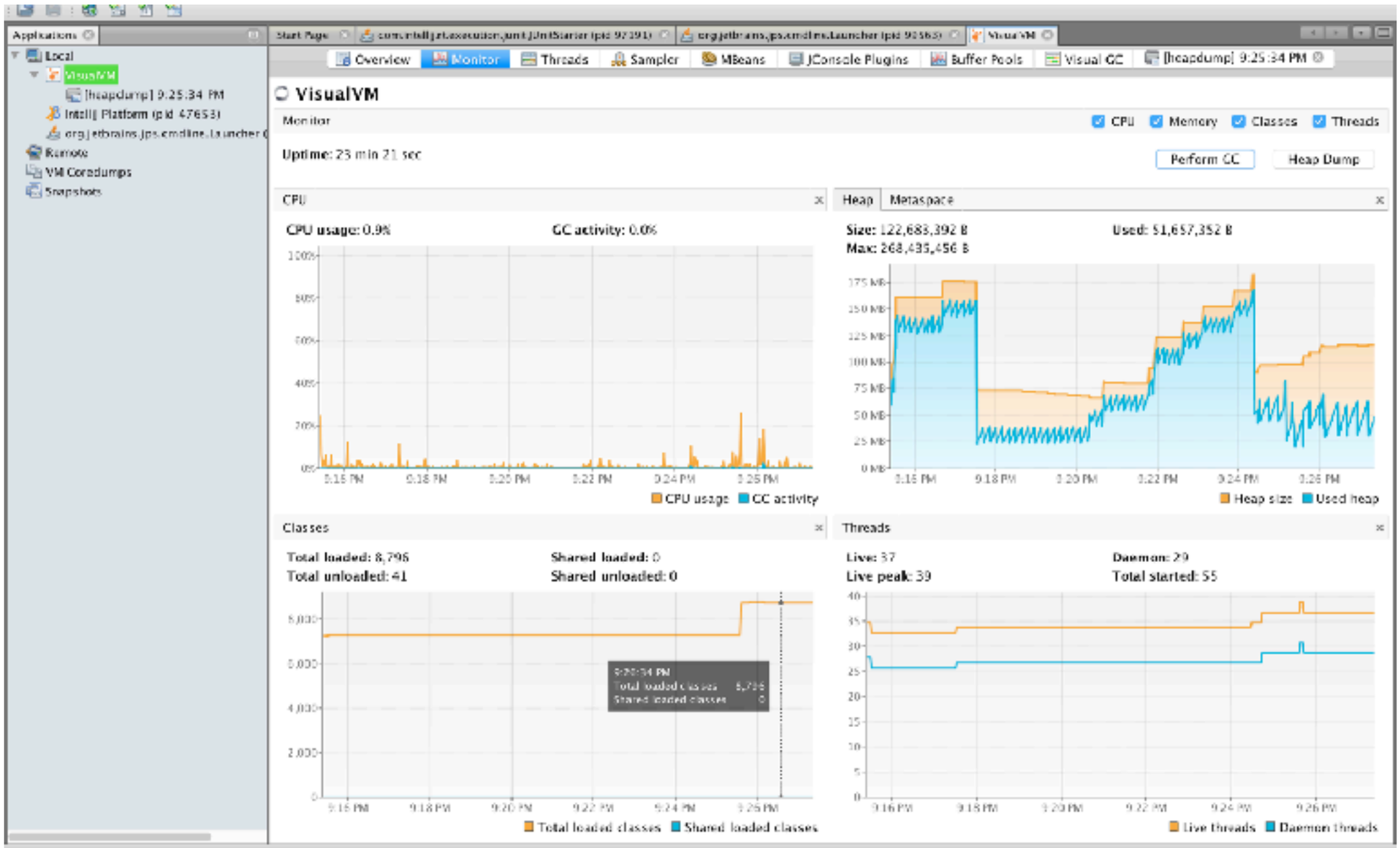
Как выглядит java.lang.OutOfMemoryError: Java heap space



Спустя 2 слайда

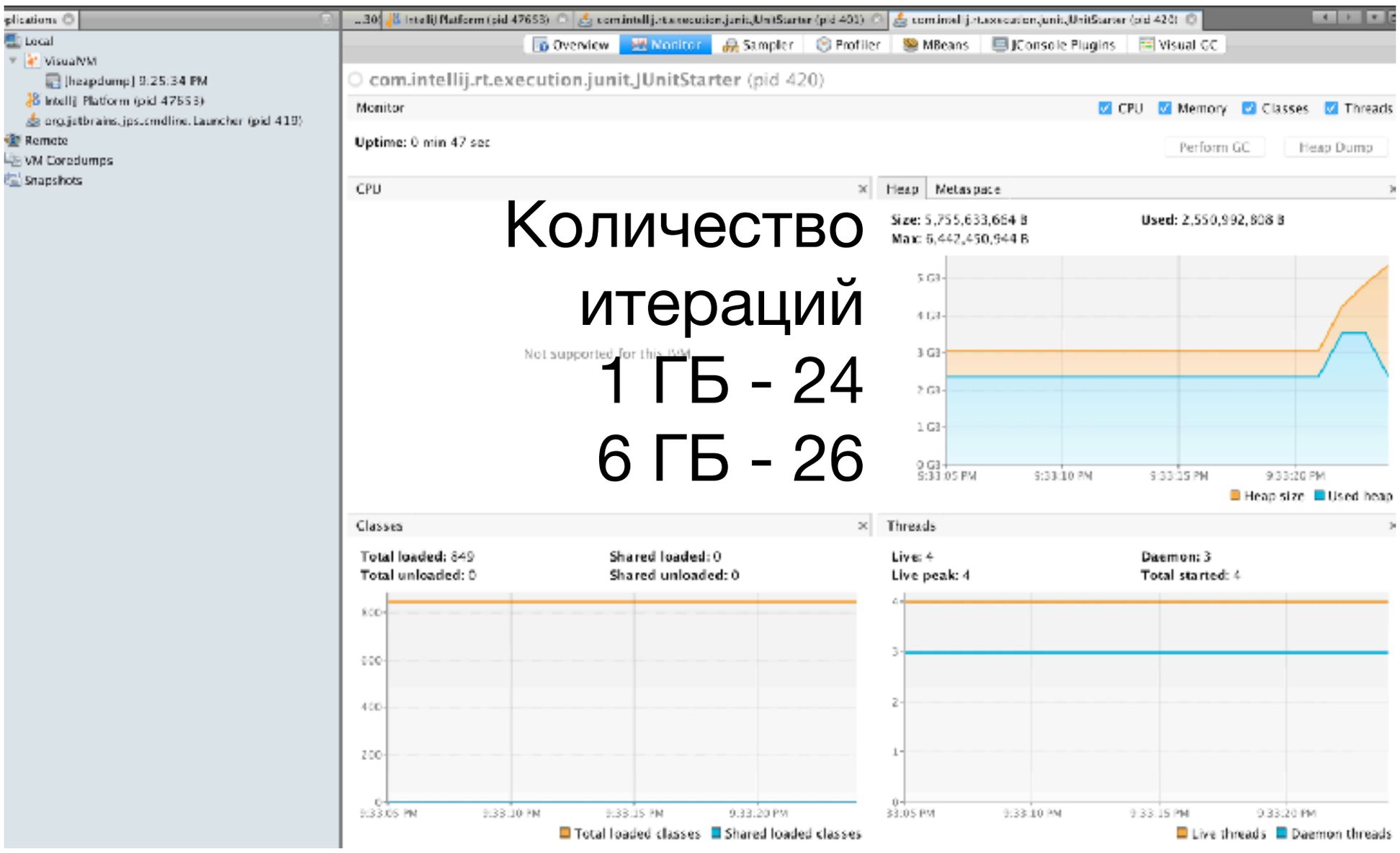


Подождем еще чуток. На 25 минуте почистила GC



Увеличим размер хипа до … 6 гигов

Добавим в VM options -Xmx6g

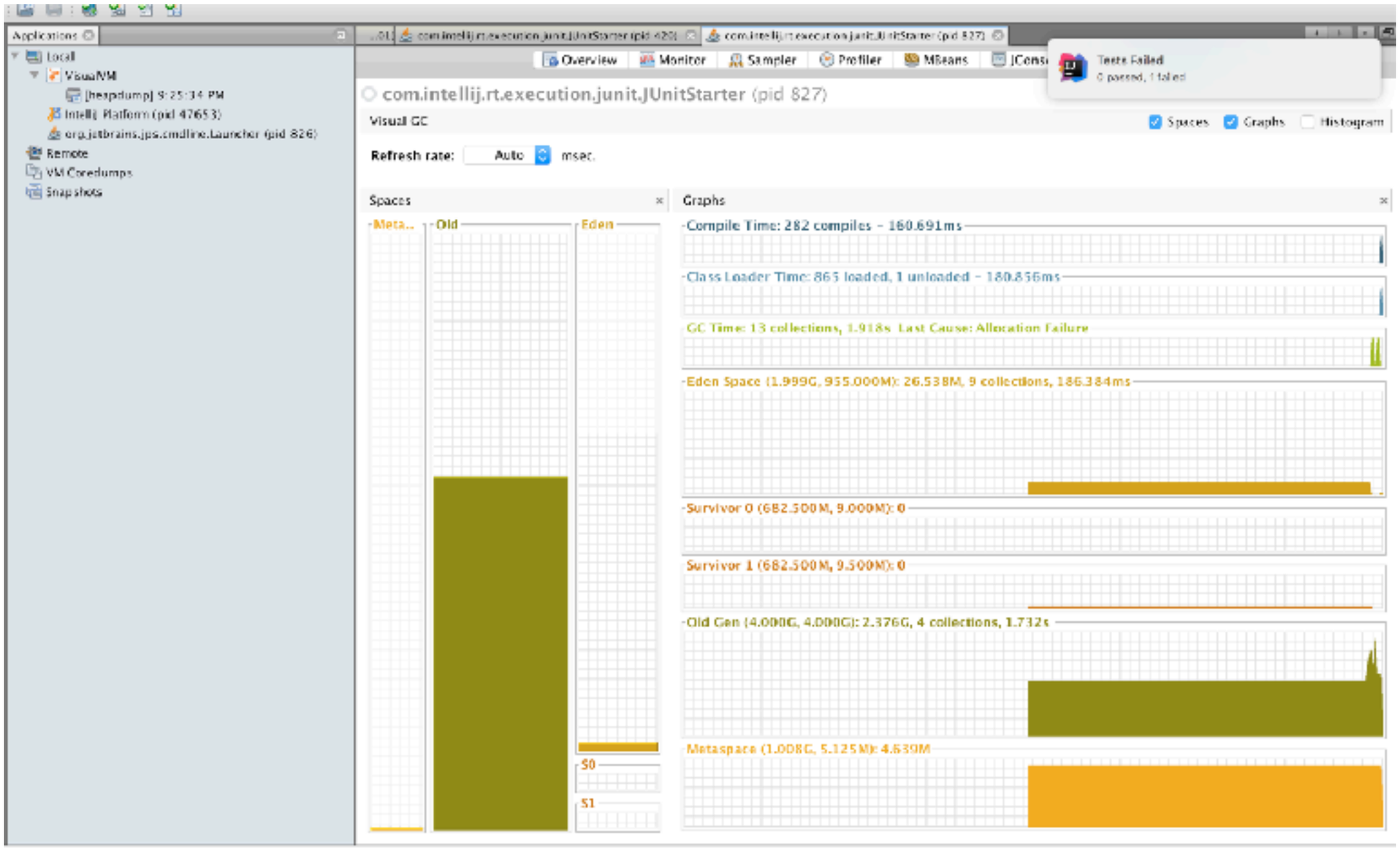


Количество итераций

1 ГБ - 24

6 ГБ - 26

Дебаг-пауза и последние вздохи перед OOM Error в GC



Отчет GC

"GC task thread#5 (ParallelGC)" os_prio=31 tid=0x00007fd78b803800 nid=0x2f03 runnable

"GC task thread#6 (ParallelGC)" os_prio=31 tid=0x00007fd78c004800 nid=0x3103 runnable

"GC task thread#7 (ParallelGC)" os_prio=31 tid=0x00007fd78c005000 nid=0x3303 runnable

"VM Periodic Task Thread" os_prio=31 tid=0x00007fd78d03f000 nid=0x5b03 waiting on condition

JNI global references: 2452

Heap

PSYoungGen total 337920K, used 0K [0x00000007aab00000, 0x00000007bff00000, 0x00000007c0000000)

eden space 327168K, 0% used [0x00000007aab00000,0x00000007aab00000,0x00000007bea80000)

from space 10752K, 0% used [0x00000007bea80000,0x00000007bea80000,0x00000007bf500000)

to space 10240K, 0% used [0x00000007bf500000,0x00000007bf500000,0x00000007bff00000)

ParOldGen total 699392K, used 623359K [0x0000000780000000, 0x00000007aab00000, 0x00000007aab00000)

object space 699392K, 89% used [0x0000000780000000,0x00000007a60bfed8,0x00000007aab00000)

Metaspace used 4756K, capacity 5152K, committed 5248K, reserved 1056768K

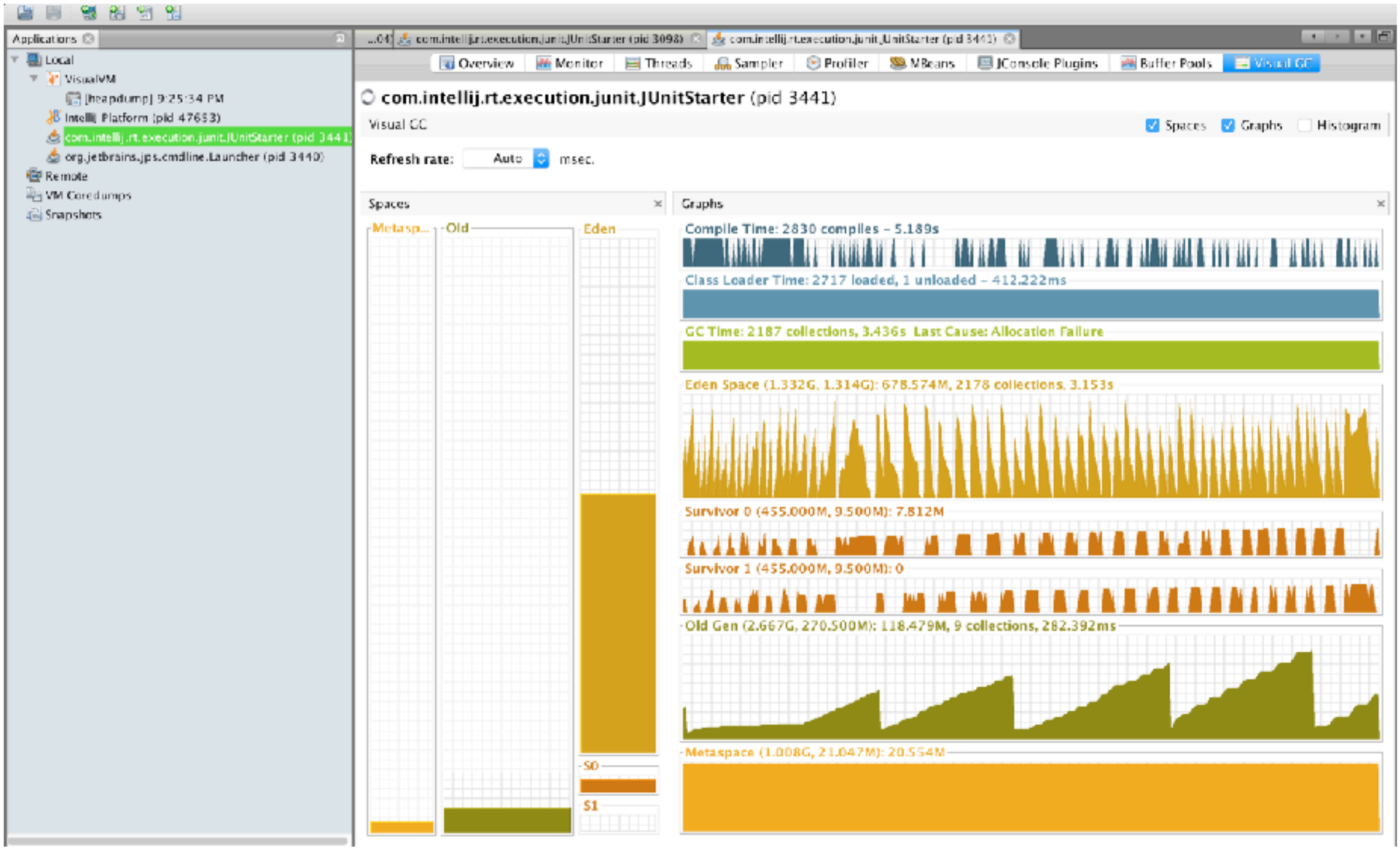
class space used 551K, capacity 592K, committed 640K, reserved 1048576K

Вопрос

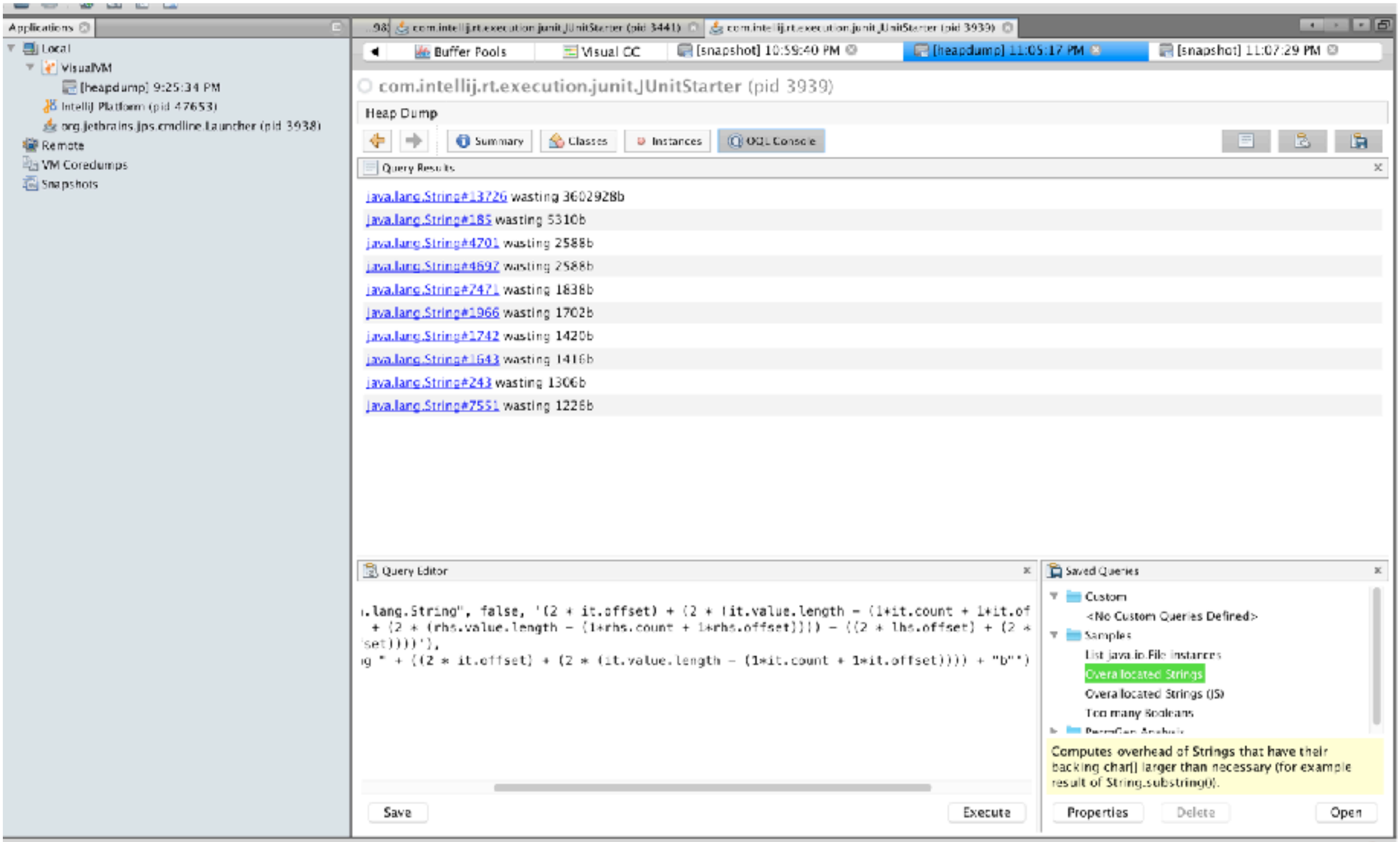
Что будет с таким кодом?

```
@Test
public void eternalCycleWithObject() {
    MyString a = new MyString("Very long
string");
    int j = 1;
    while (true) {
        int i = j + 1;
        a.setMyString(a.getMyString() + i);
        j = i;
    }
}
```

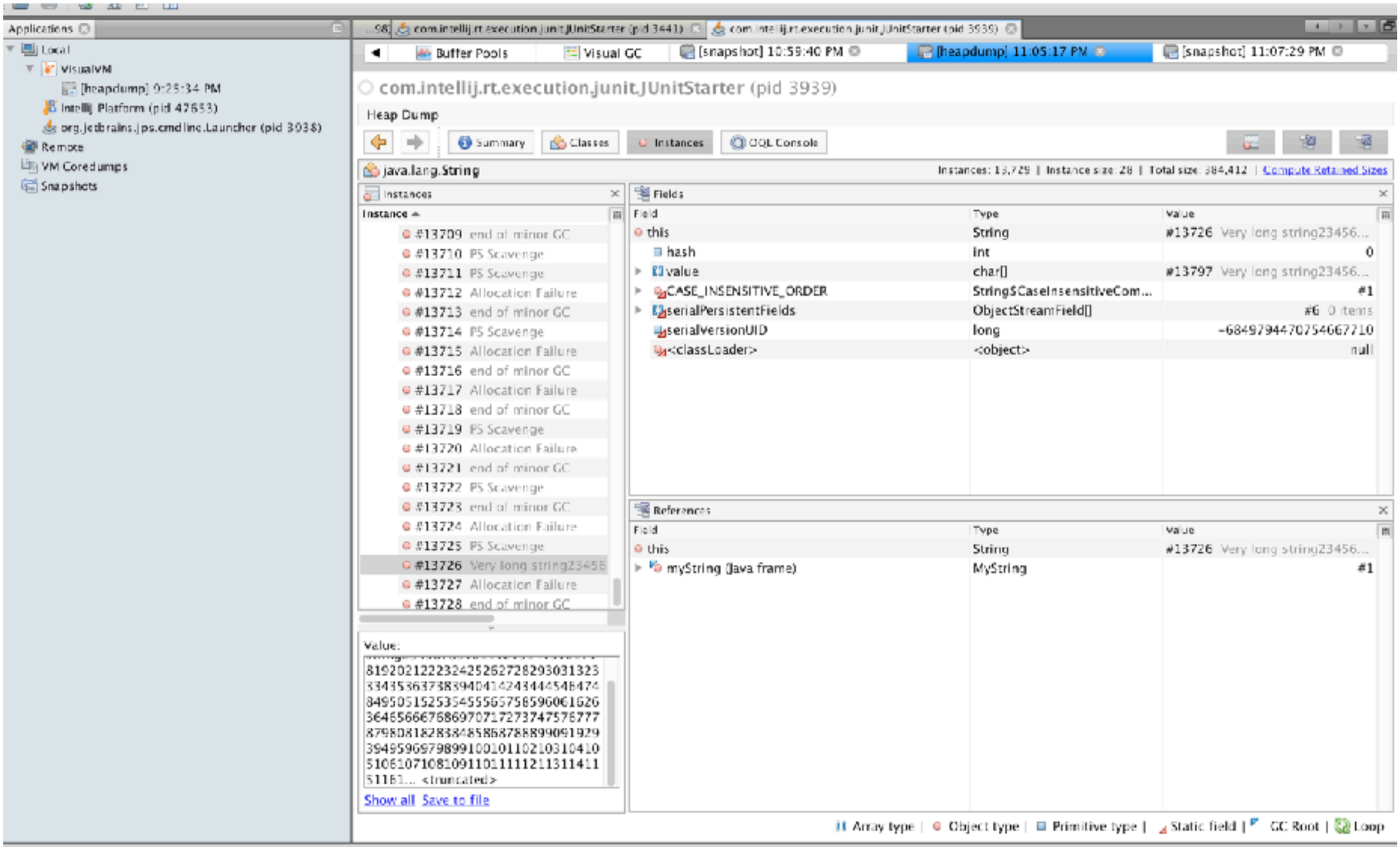
Ответ


Applications
Local
VisualVM
[heapdump] 9:25:34 PM
IntelliJ Platform (pid 47653)
com.intellij.rt.execution.junit.JUnitStarter (pid 3441)
org.jetbrains.jps.cmdline.Launcher (pid 3440)
Remote
VM Coredumps
Snapshots
com.intellij.rt.execution.junit.JUnitStarter (pid 3098)
com.intellij.rt.execution.junit.JUnitStarter (pid 3441)
Overview
Monitor
Threads
Sampler
Profiler
MBeans
JConsole Plugins
Buffer Pools
Visual GC
com.intellij.rt.execution.junit.JUnitStarter (pid 3441)
Visual GC
Spaces
Graphs
Histogram
Refresh rate:
Auto
msec.
Spaces
Metasp...
Old
Eden
S0
S1
Graphs
Compile Time: 2830 compiles – 5.189s
Class Loader Time: 2717 loaded, 1 unloaded – 412.222ms
GC Time: 2187 collections, 3.436s Last Cause: Allocation Failure
Eden Space (1.332G, 1.314G): 678.574M, 2178 collections, 3.153s
Survivor 0 (455.000M, 9.500M): 7.812M
Survivor 1 (455.000M, 9.500M): 0
Old Gen (2.667G, 270.500M): 118.479M, 9 collections, 282.392ms
Metaspace (1.008G, 21.047M): 20.554M

В хип дампе можно искать причину всех бед



Вот наше творение



Выводы

- Стек - последовательность выполнения программы, примитивы и ссылки на объекты, принадлежит потоку
- Хип или куча - объекты и массивы
- Нон-хип - методы, конструкторы и их код, поля + пул стрингов

Ресурсы

- http://www.myshared.ru/slide/1296962/ - расчет чисел в разрядной сетке
- https://habrahabr.ru/post/134102/ - сколько места занимают объекты
- https://github.com/matlux/work-kit/tree/master/articles/java - шпаргалка по всему и не только Java
- http://javadevblog.com/chto-takoe-heap-i-stack-pamyat-v-java.html - последовательность работы программы в памяти
- http://blog.jamesdbloom.com/JVMInternals.html - о структуре памяти
- https://habrahabr.ru/post/84165/ - структура хипа

GC

- https://habrahabr.ru/post/112676/
- https://www.ibm.com/developerworks/java/library/j-jtp01274/
- https://www.ibm.com/developerworks/java/library/j-jtp11253/
- https://www.ibm.com/developerworks/java/library/j-jtp10283/