

Память в Java. Garbage collector

для JiJa
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	$\pm 1.4\text{E-}45$ to $\pm 3.4028235\text{E+}38$
double	IEEE 754 floating point	0.0	64 bits	$\pm 4.9\text{E-}324$ to $\pm 1.7976931348623157\text{E+}308$

Пример: $\text{int} \rightarrow 2^{32} / 2 = 2147483648$

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.

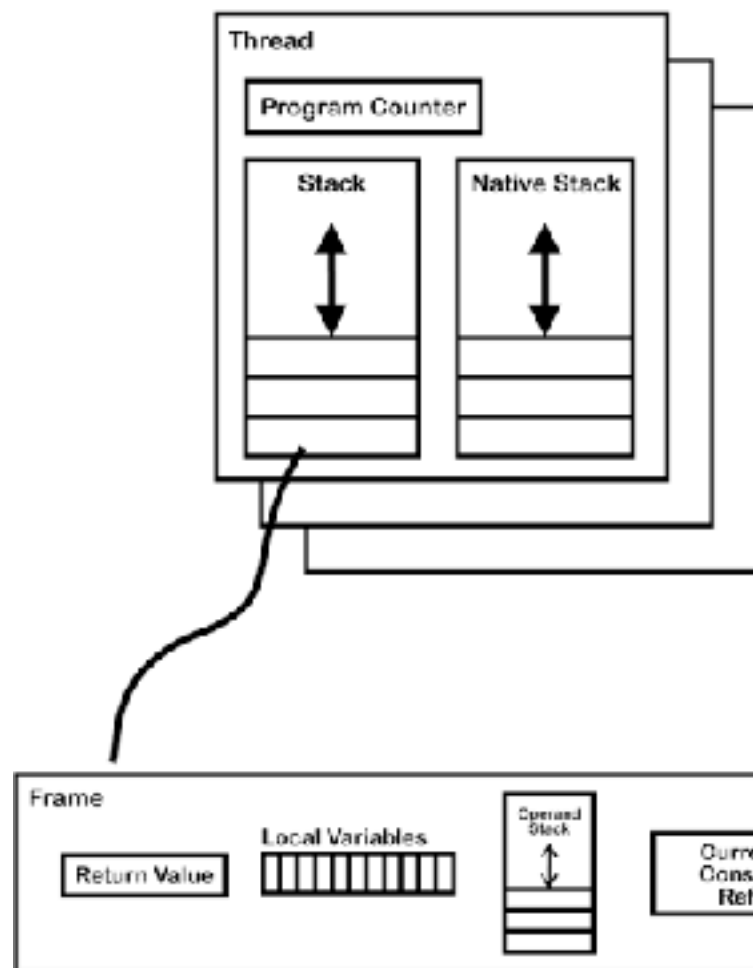
Ситуация в C++

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	<div>64-bit systems</div> Microsoft Win64 (X64 / IA64)
16	32	64	64	64	LP64 or I32LP64	
16	64	64	64	64	ILP64	
64	64	64	64	64	SILP64	
						<div>64-bit systems</div> Most Unix systems (Linux, Solaris, DEC OSF/1 Alpha, SGI Irix, HP UX 11)
						HAL; logical analog of ILP32
						UNICOS

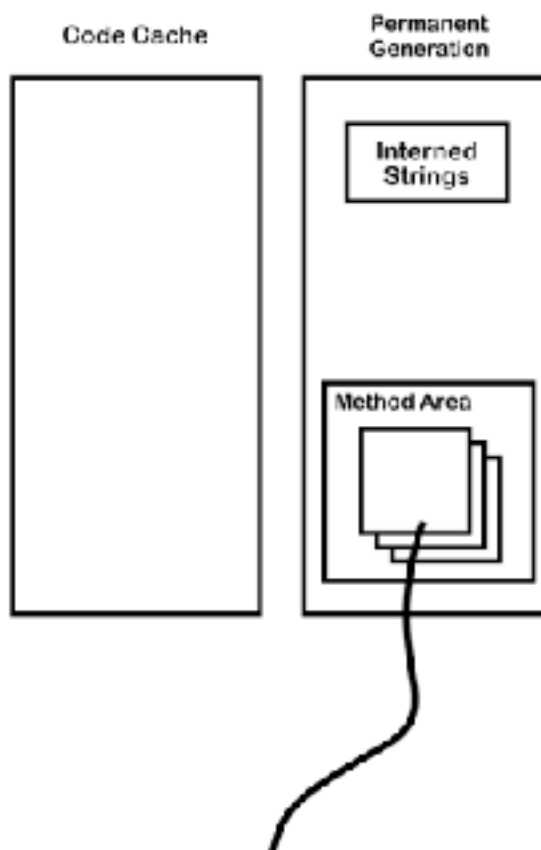
Для совместимости 64-битная система Windows поддерживает исполнение 32-битных программ, которые работают в режиме модели данных **ILP32LL**. В коде программ **необходимо учитывать разрядность** используемых данных.

Структура памяти

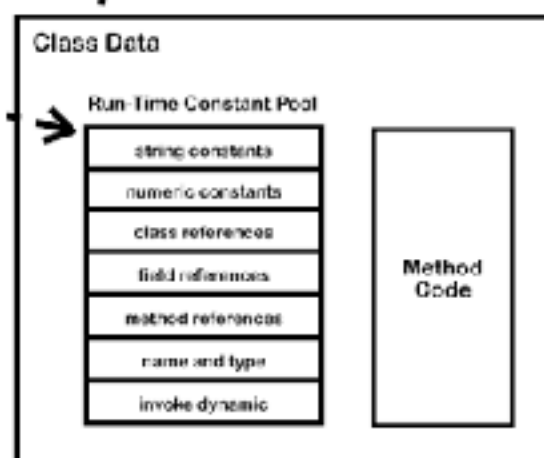
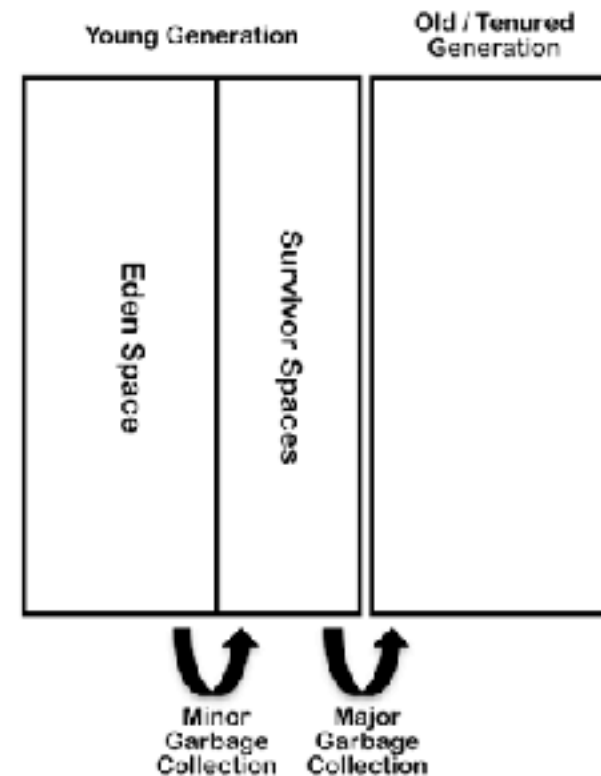
Stack



Non Heap



Heap



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

SimpleClass()

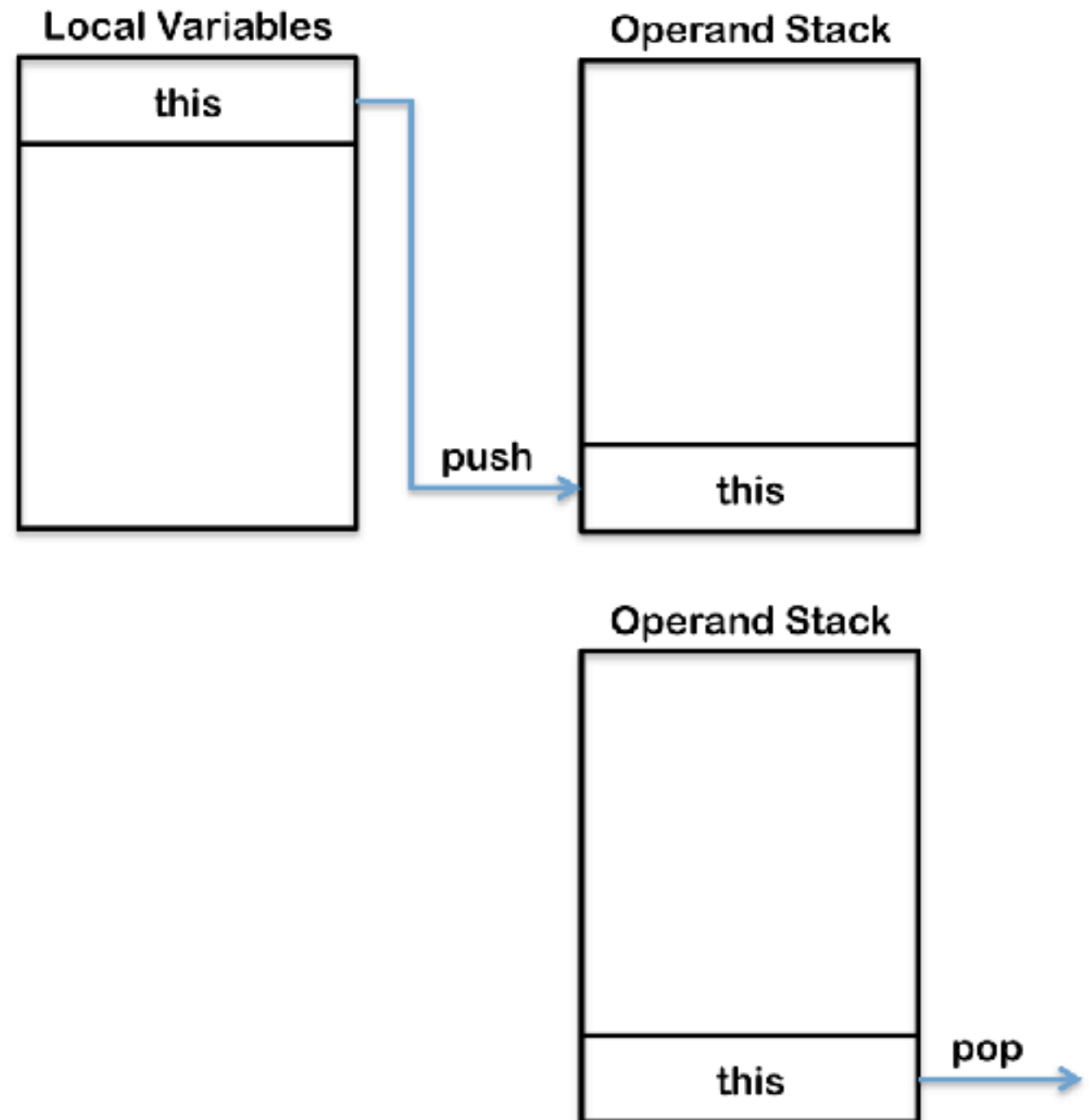
0: aload_0

Команды стека:

aload: Загружает ссылку на объект в стек операндов.

invokespecial: вызов метода инициализации, приватного метода и метода суперкласса объекта.

1: invokespecial #1



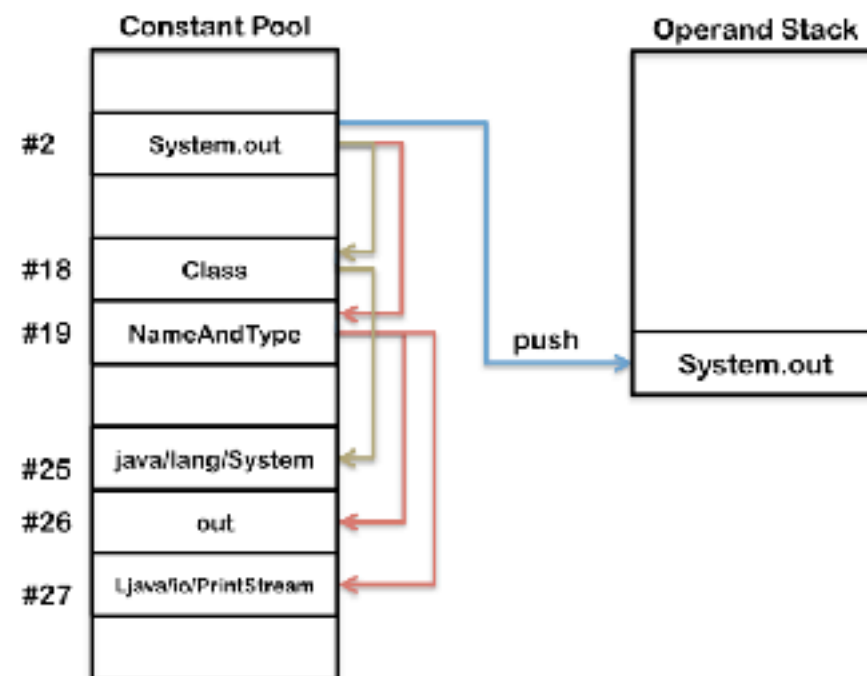
getstatic - загрузить статическую переменную из run time constant pool в operand stack.

ldc - загрузить константу в operand stack.

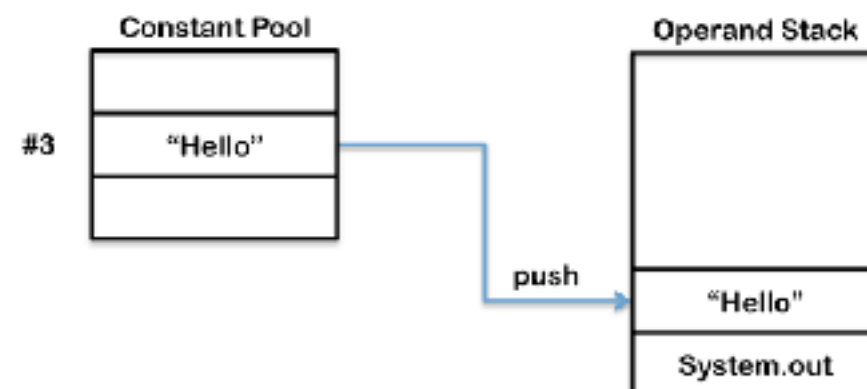
invokevirtual - вызов метода, базирующегося на классе объекта

sayHello()

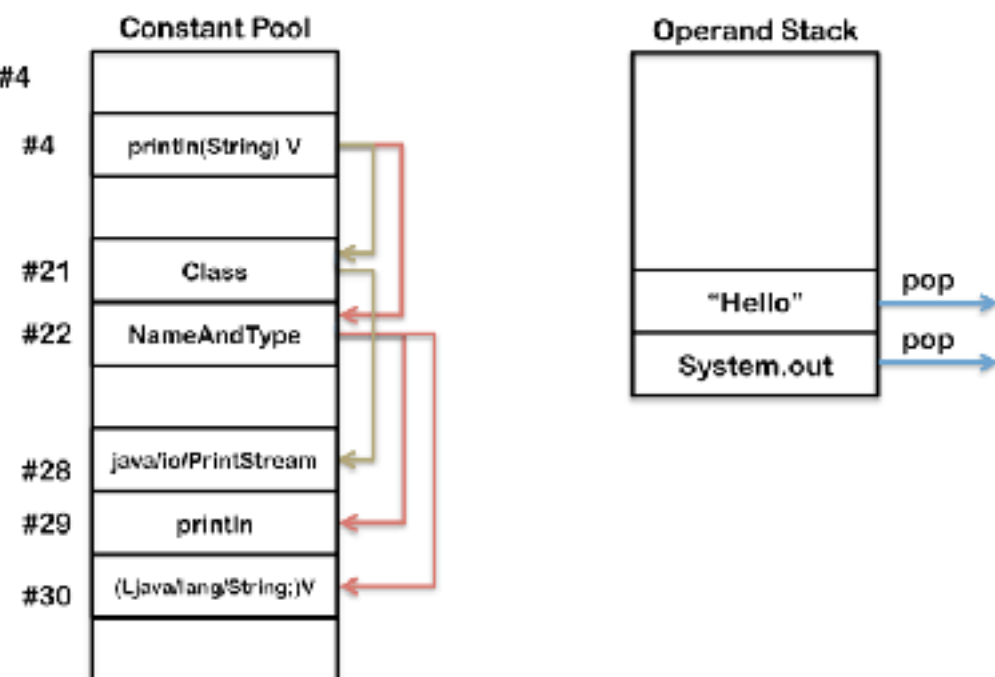
0: getstatic



3: ldc



5: invokevirtual #4



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

Heap Space						Method Area		Native Area					
Young Generation				Old Generation		Permanent Generation		Code Cache					
Virtual	From Survivor 0	To Survivor 1	Eden	Tenured	Virtual	Runtime Constant Pool	Virtual	Thread 1..N			Compile	Native	Virtual
						Field & Method Data		PC	Stack	Native Stack			
						Code							

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.

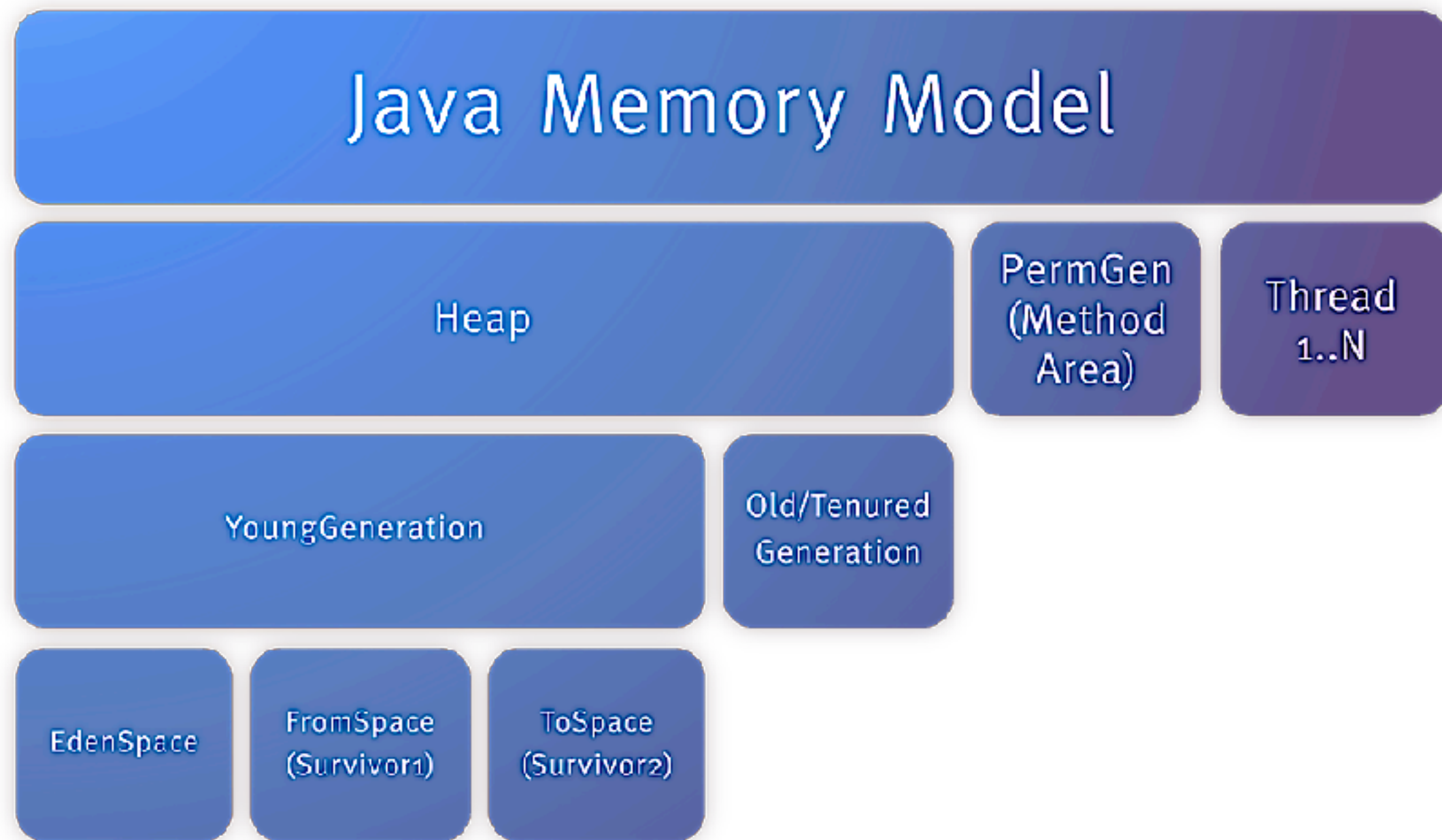
Survivor и Tenured Space (heap)

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

Heap

- Shared among all threads (global)
- Stores all created objects and arrays

Method Area

- Shared among all threads (global)
- Stores class structures like field and method data, code for methods and constructors, etc.

Thread 1..N

- Private Stack (local)
- Holds references to objects in the heap
- Stores local variables of primitive types

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

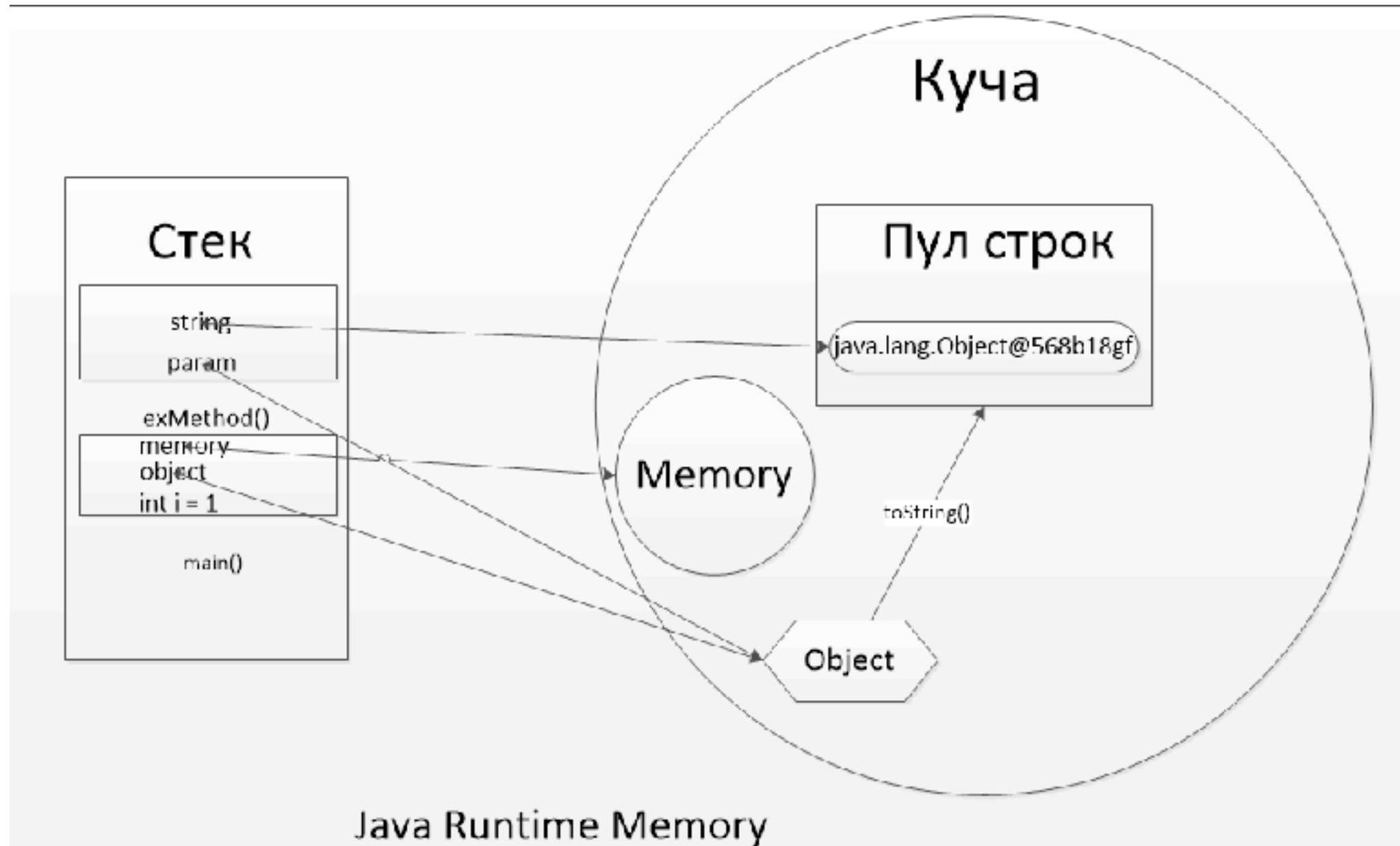
1. Все необходимые для запуска программы классы загружаются в память.
2. Команды-действия-строки записываются в стек в порядке выполнения.
3. В строке 2 создается int'овая переменная, которая хранится в памяти стека метода `main()`.
4. В куче создается объект в строке 3. Стековая память содержит ссылку 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.

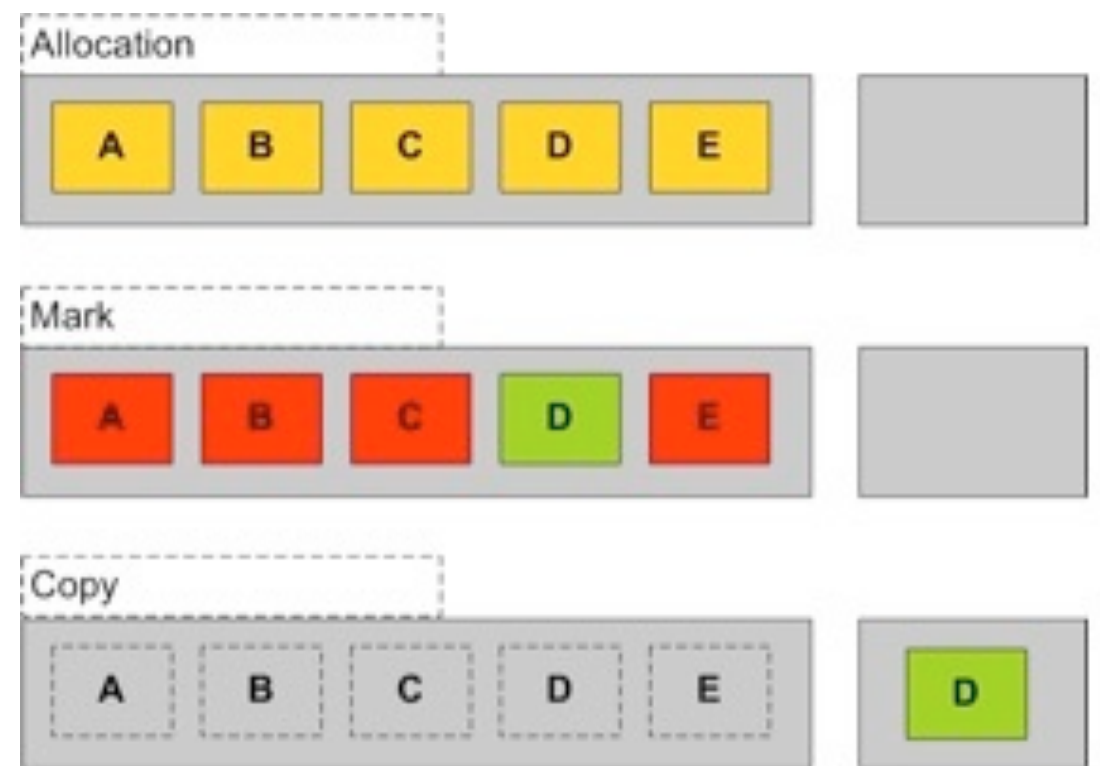
Стек хранит последовательность

Куча - объекты



GC. CopyCollection

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

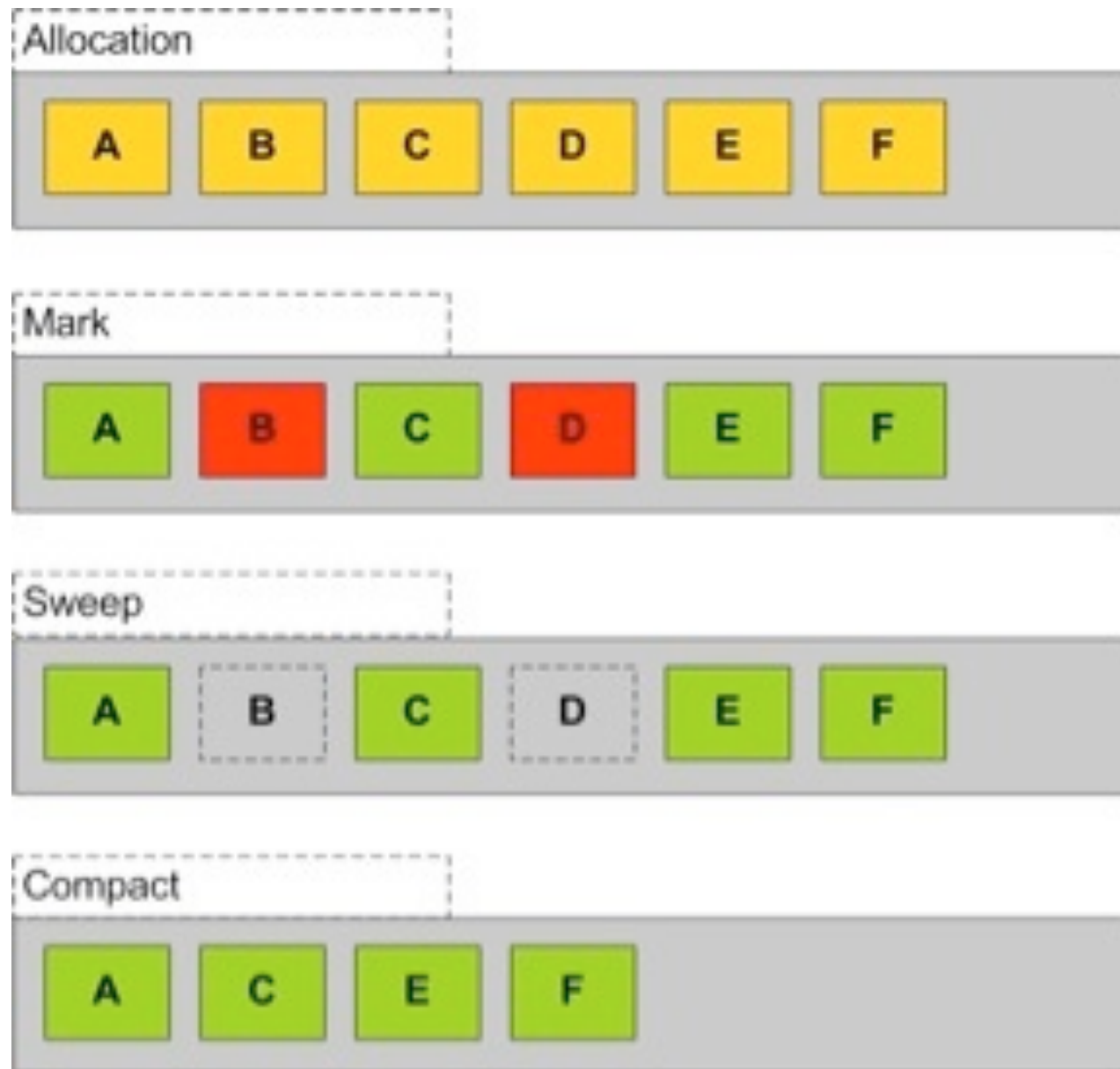


GC.Mark-Sweep-Compact Collection

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 происходит с JVM во время ее работы.
- Запустить: в консоли набрать `jvisualvm`
Путь к программе: `Path_to_Java/bin/jvisualvm`

VisualVM Overview

The screenshot displays the VisualVM application window. The top toolbar includes buttons for Overview, Monitor, Threads, Sampler, MBeans, Buffer Pools, JConsole Plugins, Visual GC, and two snapshot buttons. The main content area is titled 'VisualVM' and shows the 'Overview' tab. It lists the following information:

- PID:** 92063
- Host:** localhost
- Main class:** org.netbeans.Main
- Arguments:** --cachedir /Users/olexandra/Library/Caches/VisualVM/8u131 --userdir /Users/olexandra/Library/Application Support/VisualVM/8u131 --branding visualvm
- JVM:** Java HotSpot(TM) 64-Bit Server VM (25.152-b16, mixed mode)
- Java:** version 1.8.0_152, vendor Oracle Corporation
- Java Home:** /Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre
- JVM Flags:** <none>
- Heap dump on OOME:** enabled

Below this information, there are tabs for 'Saved data', 'JVM arguments', and 'System properties'. The 'JVM arguments' tab is selected, showing a list of arguments:

- Djdk.home=/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home
- Dnetbeans.default_userdir_root=/Users/olexandra/Library/Application Support/VisualVM
- Dnetbeans.dirs=/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/visualvm:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/visualvm
- Dnetbeans.home=/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/visualvm/platform
- Xms24m
- Xmx256m
- Dsun.jvmstat.perdata.syncWaitMs=10000
- Dsun.java2d.noddraw=true
- Dsun.java2d.d3d=false
- Dnetbeans.keyring.no.master=true
- Dplugin.manager.install.global=false
- add-exports=java.desktop/sun.awt=ALL-UNNAMED
- add-exports=jdk.jvmstat/sun.jvmstat.monitor.event=ALL-UNNAMED
- add-exports=jdk.jvmstat/sun.jvmstat.monitor=ALL-UNNAMED
- add-exports=java.desktop/sun.swing=ALL-UNNAMED
- add-exports=jdk.attach/sun.tools.attach=ALL-UNNAMED
- add-modules=java.activation
- XX:+IgnoreUnrecognizedVMOptions
- XX:+HeapDumpOnOutOfMemoryError
- XX:HeapDumpPath=/Users/olexandra/Library/Application Support/VisualVM/8u131/var/log/heapdump.hprof

Маленькая но грозная программа

```
@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?

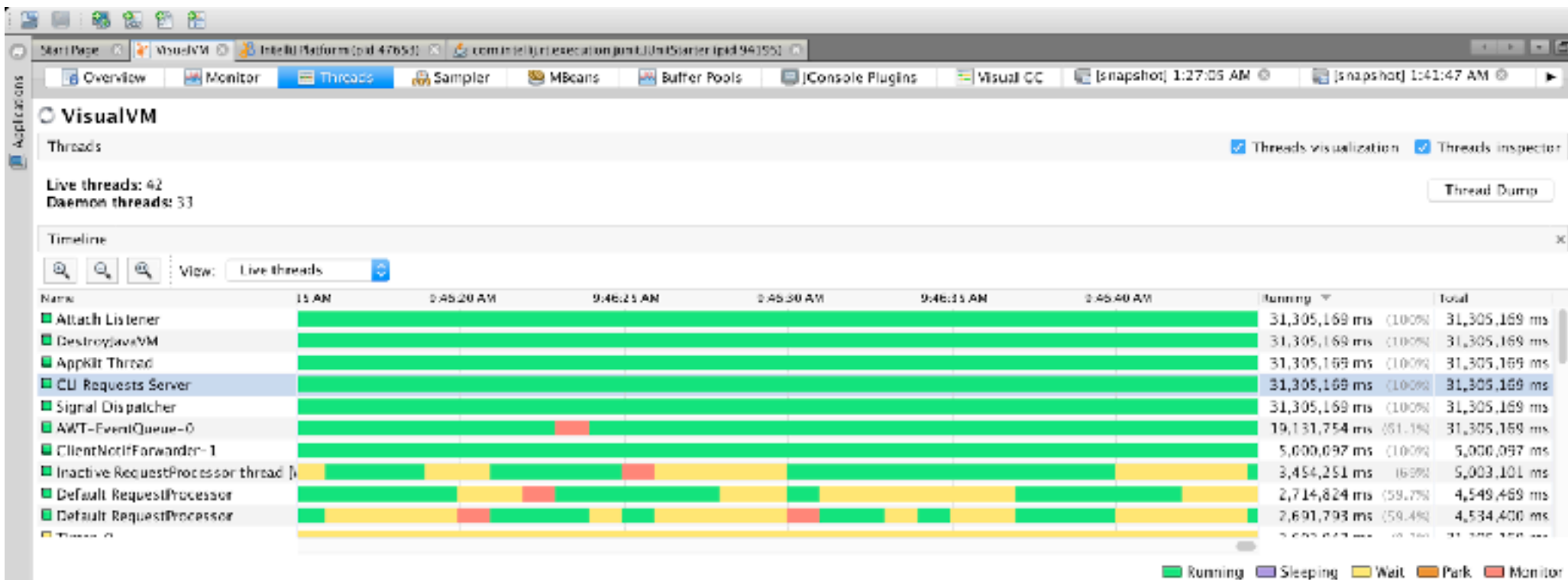
В классе `AbstractStringBuilder` есть метод, который выделяет память под новый массив символов в `String`

```
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



Threads Inspector

2017-12-04 09:47:29

"Finalizer" - Thread tg3
java.lang.Thread.State: WAITING
at java.lang.Object.wait(Native Method)
- waiting on <78e84282> (a java.lang.ref.ReferenceQueue\$Lock)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)
at java.lang.ref.Finalizer\$FinalizerThread.run(Finalizer.java:289)

Locked ownable synchronizers:
- None

"Signal Dispatcher" - Thread b04
java.lang.Thread.State: RUNNABLE
Locked ownable synchronizers:
- None

Refresh

Debug MemoryUsageTest.eternalCycle

Debugger Console

Frames Threads

"main"@1 in group "main": RUNNING

"Finalizer"@794: WAIT

"Reference Handler"@795: WAIT

"Signal Dispatcher"@793: RUNNING

VisualVM.Heap

Start Page VisualVM IntelliJ Platform (pid 47653) com.intellij.rt.execution.junit.JUnit4TestRunner (pid 94195) Local Application (pid 96309) com.intellij.rt.execution.junit.JUnit4TestRunner (pid 97191)

Overview Monitor Threads Sampler Profiler MBeans Buffer Pools JConsole Plugins Visual GC

com.intellij.rt.execution.junit.JUnit4TestRunner (pid 97191)

Sampler ☐ Settings

Sample: ☐ CPU ☒ Memory

Status: memory sampling in progress

Heap histogram Per thread allocations

Classes: 793 Instances: 170,663 Bytes: 158,771,128

Class Name	Bytes (%)	Bytes	Instances
char[]		131,449,800 (82.7%)	10,228 (5.9%)
int[]		21,879,856 (13.7%)	3,418 (2.0%)
java.util.TreeMap\$Entry		1,472,400 (0.9%)	36,810 (21.5%)

Heap histogram Per thread allocations

Classes: 792 Instances: 370,608 Bytes: 382,888,872

Class Name	Bytes (%)	Bytes	Instances
memory.MemoryUsageTest		16 (0.0%)	1 (0.0%)
java.lang.Class		186,920 (0.1%)	1,793 (1.0%)
java.util.TreeMap\$KeySet		183,984 (0.1%)	11,499 (6.7%)

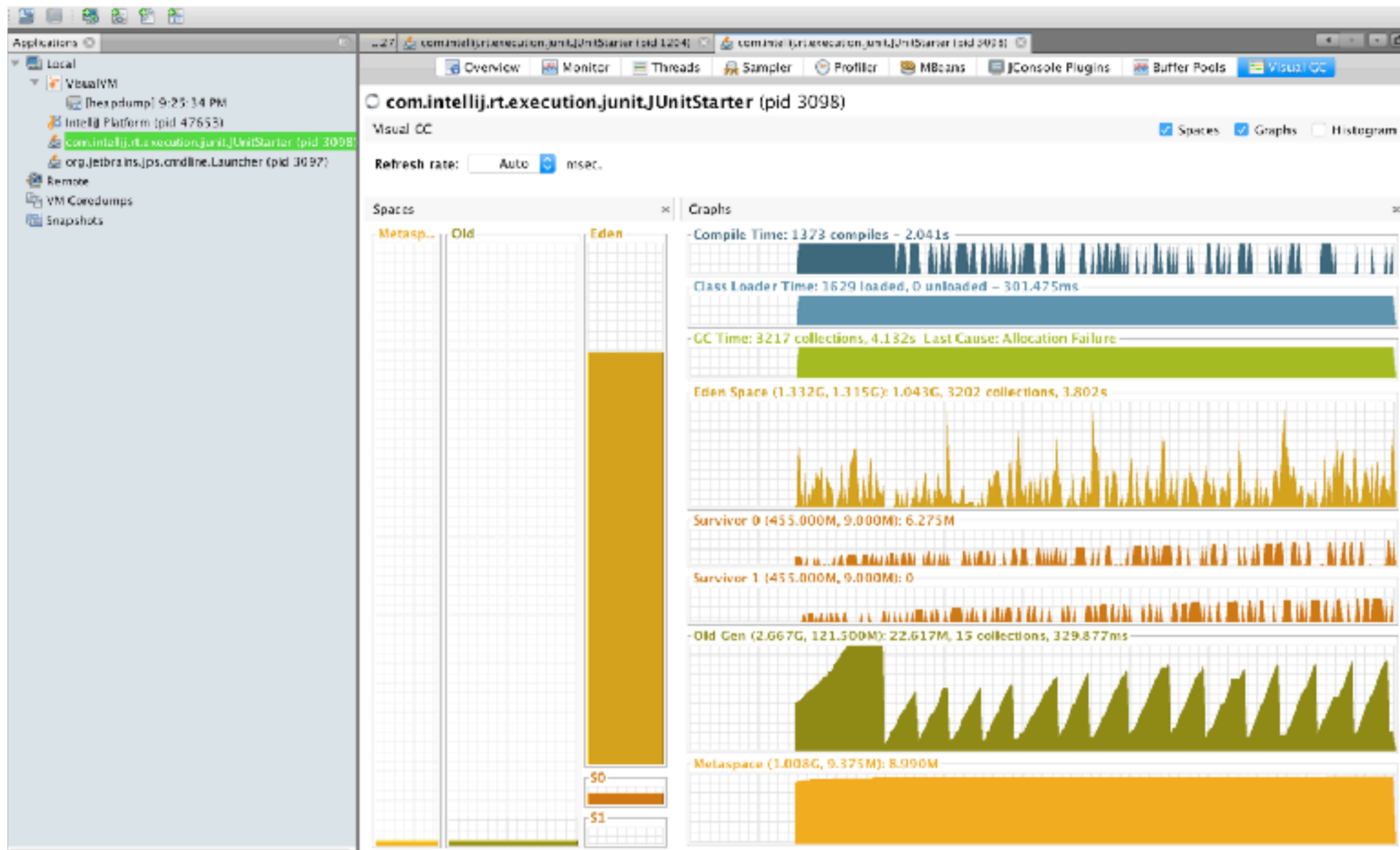
Heap histogram Per thread allocations

Classes: 873 Instances: 336,242 Bytes: 724,108,024

Class Name	Bytes (%)	Bytes	Instances
java.lang.String		333,312 (0.0%)	13,888 (4.1%)
java.lang.String[]		72,512 (0.0%)	2,644 (0.7%)
java.lang.StringBuilder		1,512 (0.0%)	63 (0.0%)
java.security.Provider\$UString		672 (0.0%)	28 (0.0%)
java.lang.StringCoding\$StringEncoder		128 (0.0%)	4 (0.0%)
javax.management.openmbean.TabularDataSupport		21,216 (0.0%)	884 (0.5%)

Class Name Filter (Contains)

GC справляется



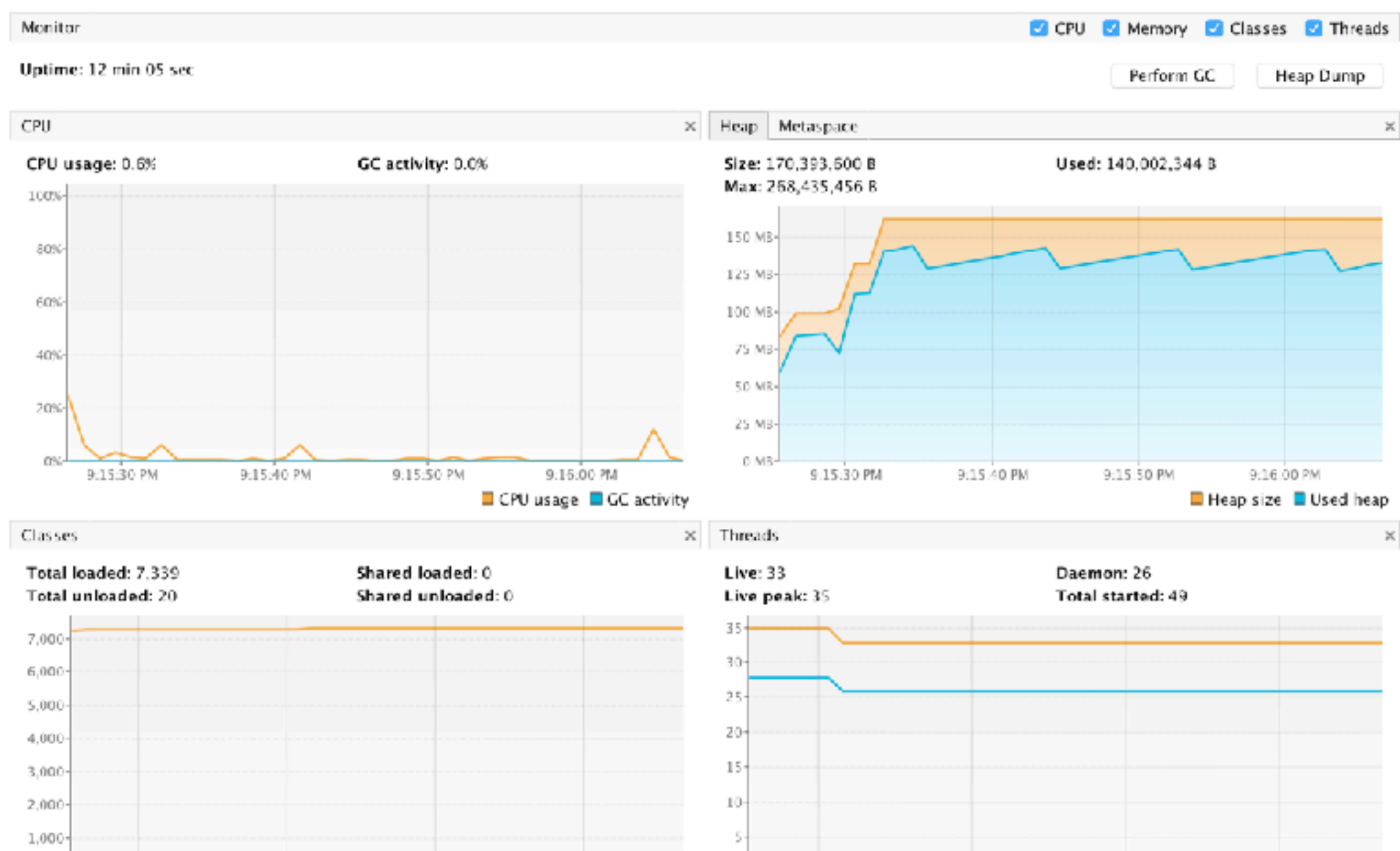
А теперь будем прибавлять строки

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

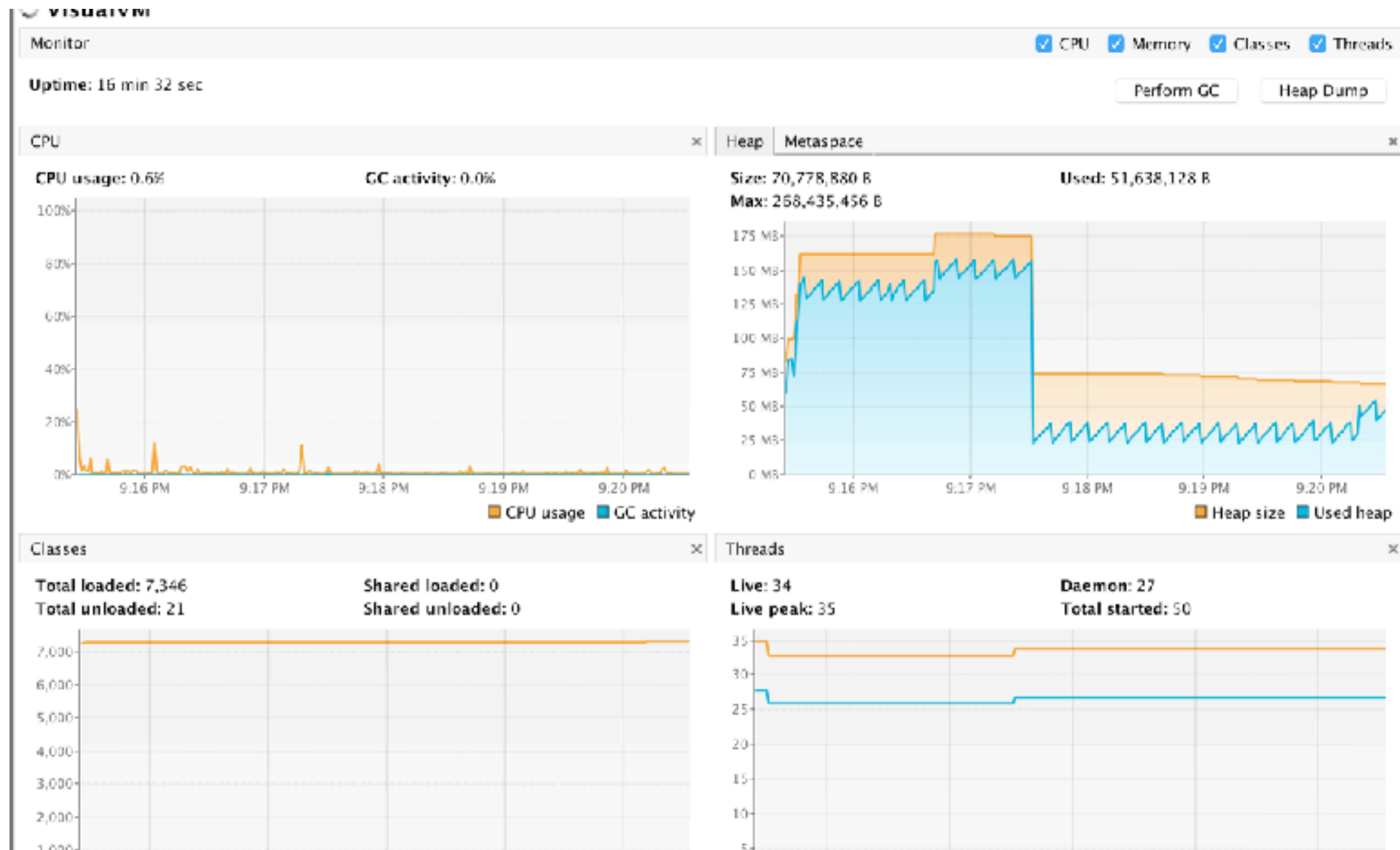
public class MyString {
    private String myString;

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

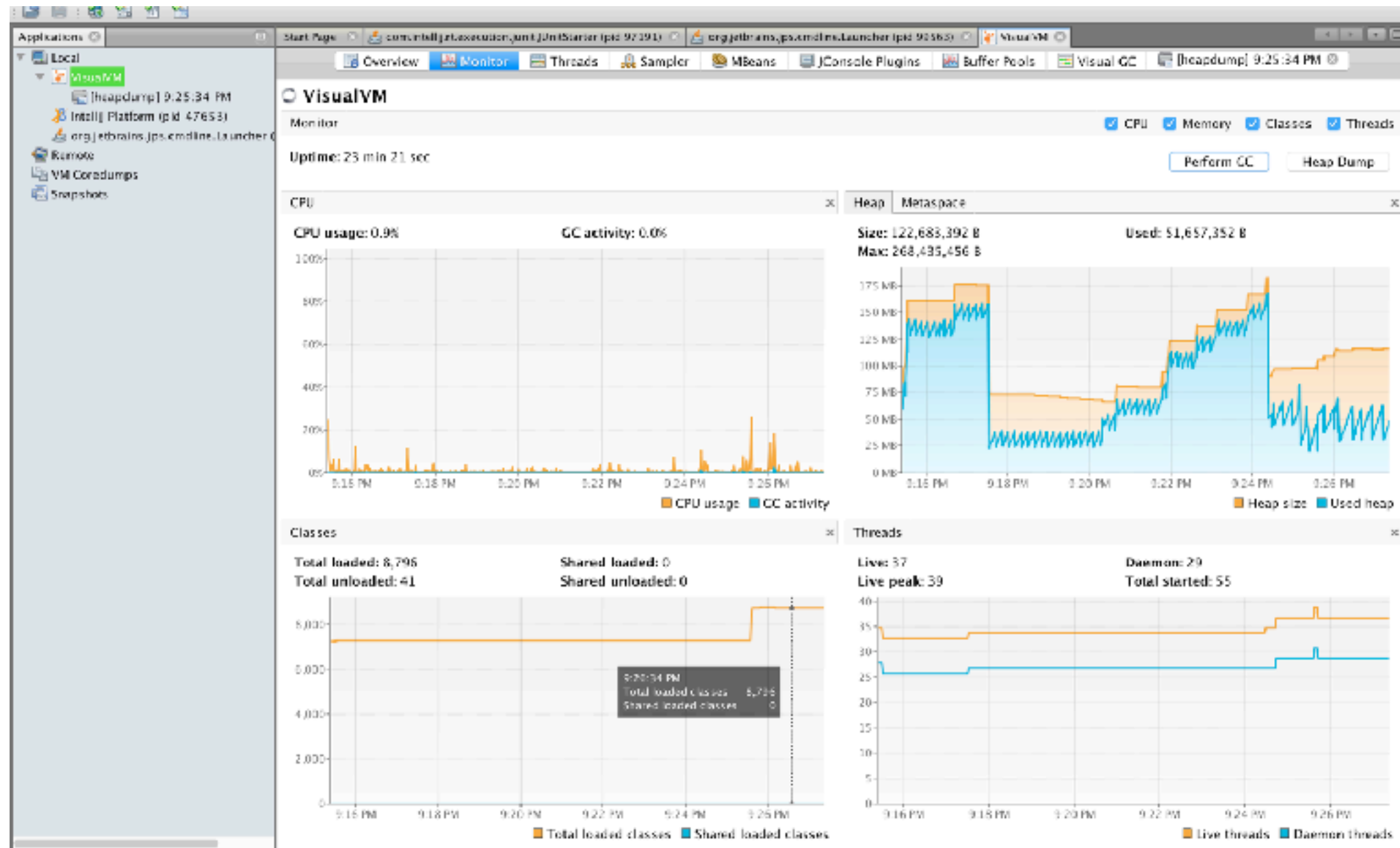

Как выглядит java.lang.OutOfMemoryError: Java heap space



Спустя 2 слайда

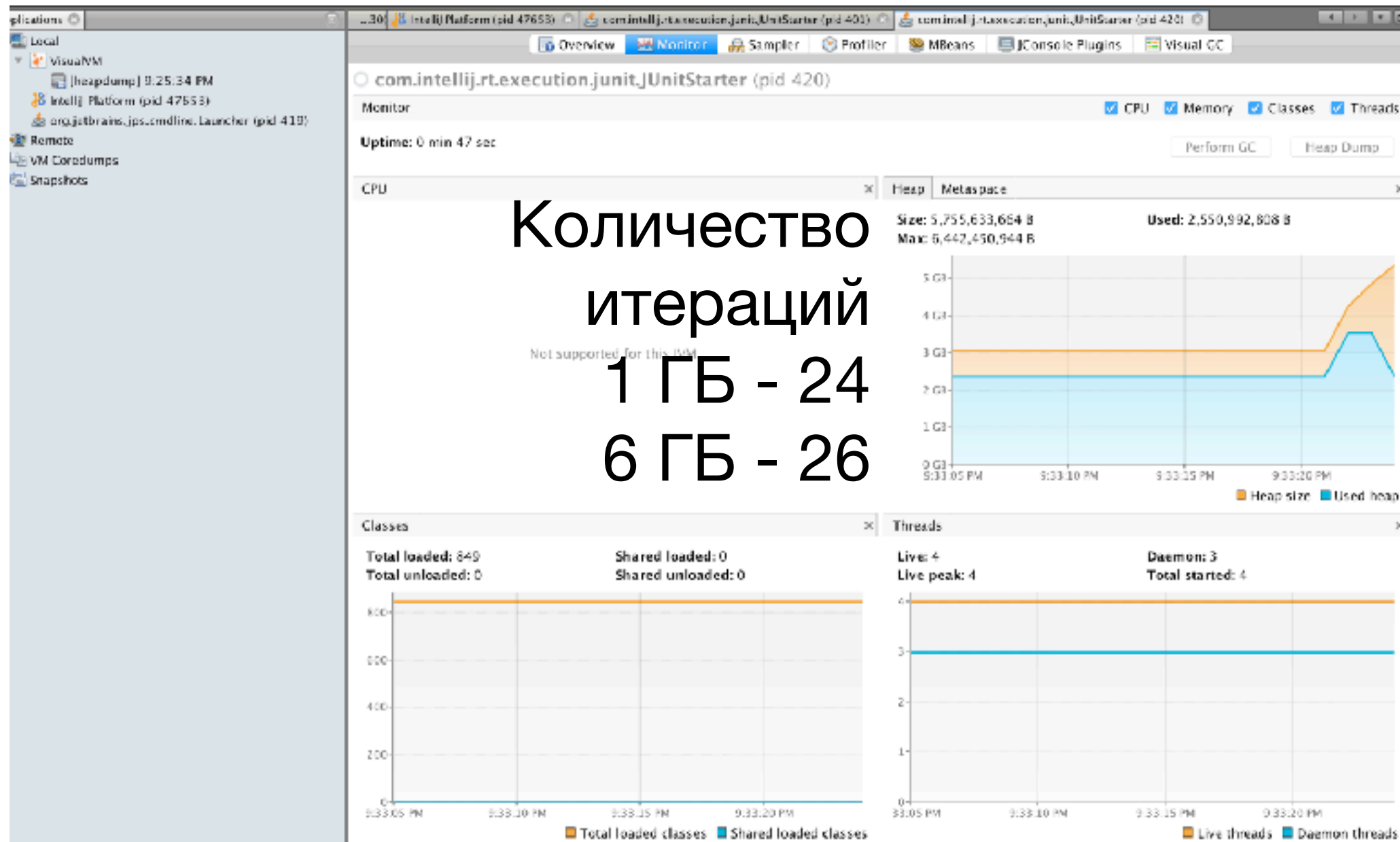


Подождем еще чуток. На 25 минуте почистила GC

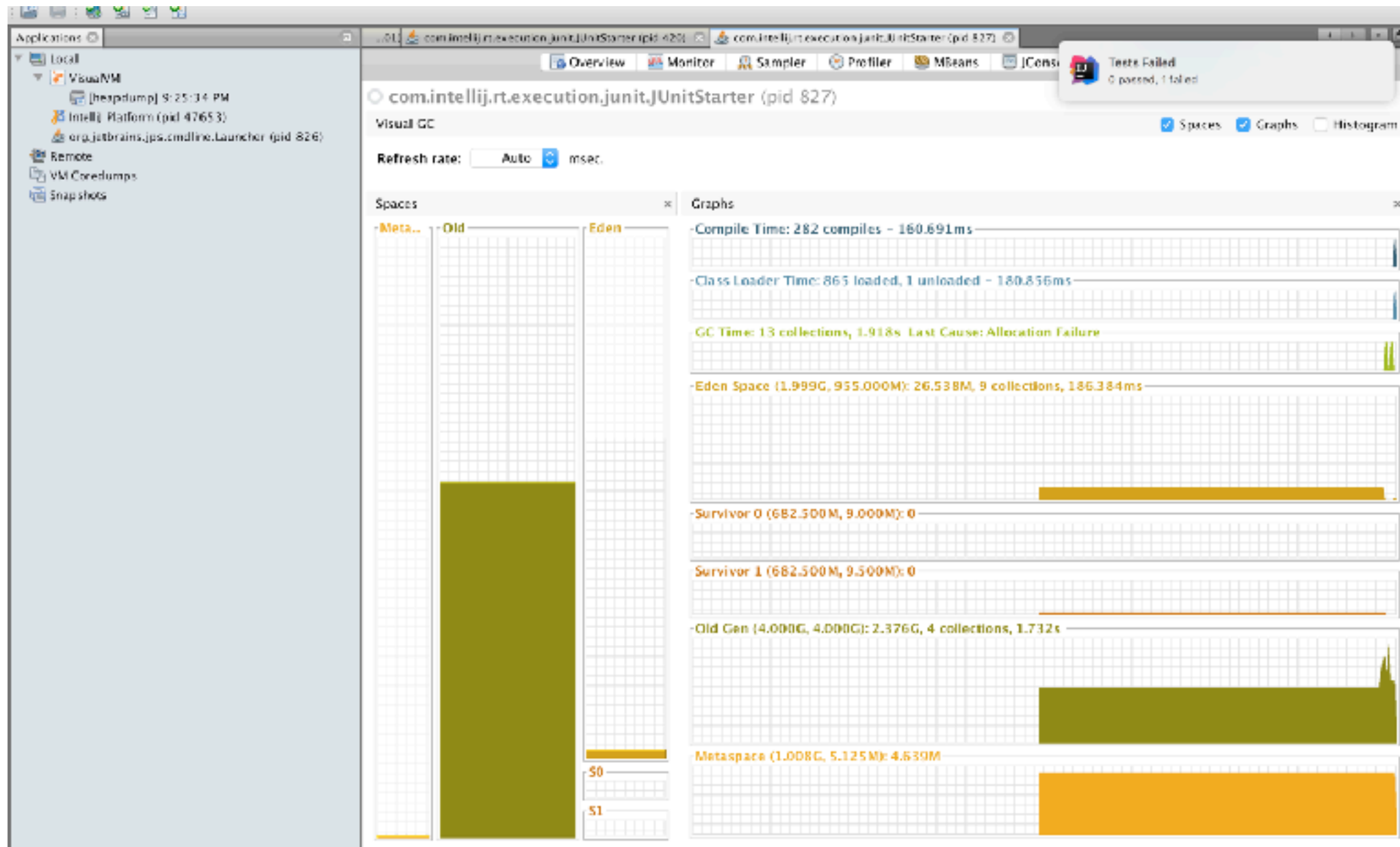


Увеличим размер хипа до ... 6 гигов

Добавим в VM options -Xmx6g



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

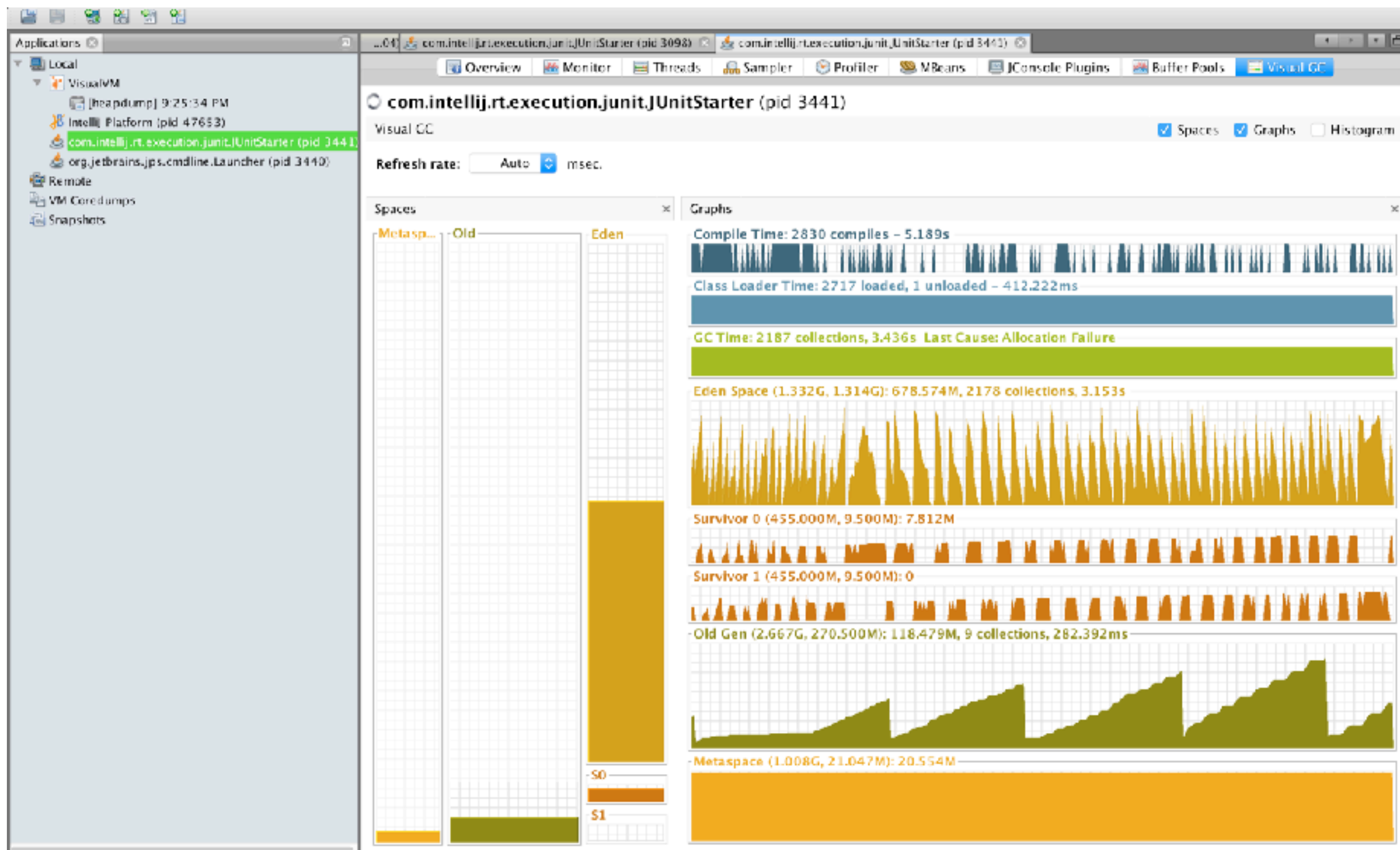
class space used 551K, capacity 592K, committed 640K, reserved 1048576K

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



В хип дампе можно искать причину всех бед

The screenshot shows the IntelliJ IDEA interface with the Heap Dump tool open. The left sidebar shows the 'Applications' tree with 'Local' > 'VisualVM' > '[heapdump] 9:25:34 PM' selected. The main panel displays the 'Heap Dump' for 'com.intellij.rt.execution.junit.JUnit4Runner (pid 3939)'. The 'Query Results' tab shows a list of Java.lang.String instances and the amount of memory they are wasting.

Object	Wasting
java.lang.String#13726	wasting 3602928b
java.lang.String#185	wasting 5310b
java.lang.String#4701	wasting 2588b
java.lang.String#4697	wasting 2588b
java.lang.String#7471	wasting 1838b
java.lang.String#1966	wasting 1702b
java.lang.String#1742	wasting 1420b
java.lang.String#1643	wasting 1416b
java.lang.String#243	wasting 1306b
java.lang.String#7551	wasting 1226b

The 'Query Editor' at the bottom shows the following query:

```
java.lang.String", false, '(2 * it.offset) + (2 * (it.value.length - (1*it.count + 1*it.offset))) - ((2 * lhs.offset) + (2 * lhs.value.length - (1*rhs.count + 1*rhs.offset))) - ((2 * lhs.offset) + (2 * lhs.value.length - (1*rhs.count + 1*rhs.offset)))',  
ig " + ((2 * it.offset) + (2 * (it.value.length - (1*it.count + 1*it.offset)))) + "b")
```

The 'Saved Queries' panel on the right shows a list of queries, with 'Over allocated Strings' highlighted. A description at the bottom states: 'Computes overhead of Strings that have their backing char[] larger than necessary (for example result of String.substring()).'

Вот наше творение

The screenshot shows the IntelliJ IDEA IDE with the Visual GC tool open. The main window displays the heap dump for the class `com.intellij.rt.execution.junit.JUnit4TestRunner` (pid 3939). The left sidebar shows the project structure with a 'VisualVM' tab selected. The main window is divided into several panes:

- Instances:** A list of memory instances for the selected class. The instance `#13726` is selected, which is a 'Very long string23456'.
- Fields:** A table showing the fields of the selected instance. The fields are: `this` (String, #13726), `hash` (int, 0), `value` (char[], #13797), `CASE_INSENSITIVE_ORDER` (String\$CaseInsensitiveCom..., #1), `serialPersistentFields` (ObjectStreamField[], #6), `serialVersionUID` (long, -6849794470754667710), and `<classLoader>` (<object>, null).
- References:** A table showing the references to the selected instance. The reference is `myString (java frame)` (MyString, #1).
- Value:** A text area showing the raw memory dump of the selected instance, displaying a long sequence of hexadecimal digits.

The bottom status bar indicates the current view is 'Array type'.

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/что-такое-heap-i-stack-память-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/>