Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Кафедра Вычислительной техники

Дисциплина: Основы программной инженерии

Лабораторная работа №4

«Мониторинг и профилирование»

Выполнил Борзых А.А, гр. P3217

Преподаватель: Клименков С.В.

Санкт-Петербург, 2016 г.

Задание

Вариант 277

1. Для своей программы из [лабораторной работы #4](http://helios.cs.ifmo.ru/disciplines/iaps#lab4) по дисциплине "Программирование интернет-приложений" реализовать:

* MBean, считающий общее число установленных пользователем точек, а также число точек, попадающих в область. В случае, если пользователь совершил 3 "промаха" подряд, разработанный MBean должен отправлять оповещение об этом событии.
* MBean, определяющий площадь получившейся фигуры.

2. С помощью утилиты JConsole провести мониторинг программы:

* Снять показания MBean-классов, разработанных в ходе выполнения задания 1.
* Определить имена всех потоков, выполняющихся при запуске программы.

3. С помощью утилиты VisualVM провести мониторинг и профилирование программы:

* Снять график изменения показаний MBean-классов, разработанных в ходе выполнения задания 1, с течением времени.
* Определить имя потока, потребляющего наибольший процент времени CPU.

4. Получить HeapDump, и с помощью утилиты VisualVM локализовать и устранить "утечку памяти" в программе ниже.

Исходный код разработанных MBean классов

Регистрация MBeans:

MBeanServer mbs = ManagementFactory.*getPlatformMBeanServer*();  
**try** {  
 ObjectName area\_mbean = **new** ObjectName(**"ifmo.sandyre.lab4:type=AreaMBean"**);  
 ObjectName shots\_mbean = **new** ObjectName(**"ifmo.sandyre.lab4:type=ShotsMBean"**);  
  
 mbs.registerMBean(draw.getAreaComputer(), area\_mbean);  
 mbs.registerMBean(draw.getShotsController(), shots\_mbean);  
} **catch**(Exception e) {  
 System.***out***.println(e.toString());  
}

ShotsController:

**public interface** ShotsControllerMBean **extends** NotificationEmitter {  
 **int** getTotalShotsCount();  
 **int** getInShotsCount();  
}

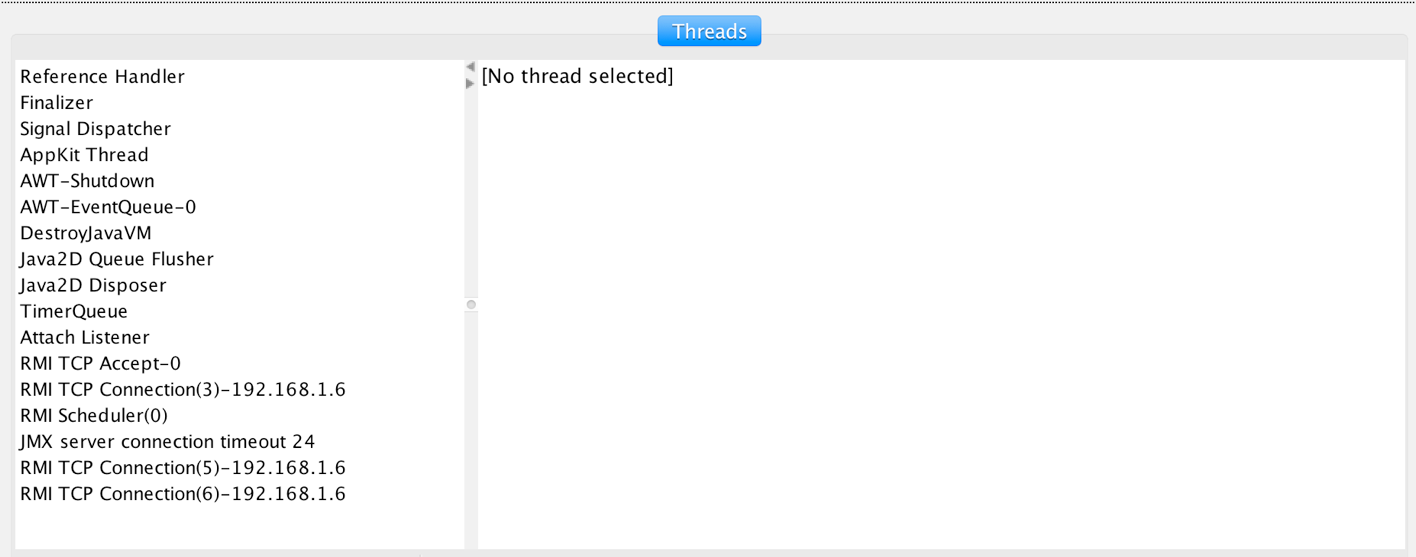
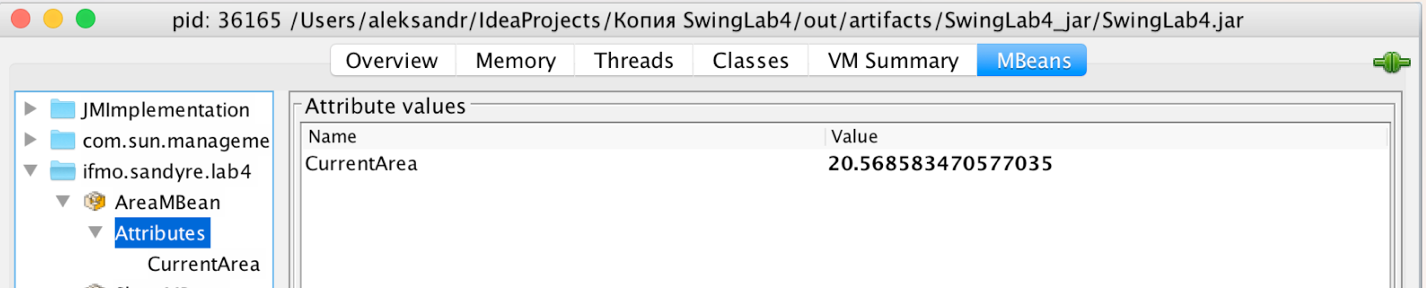
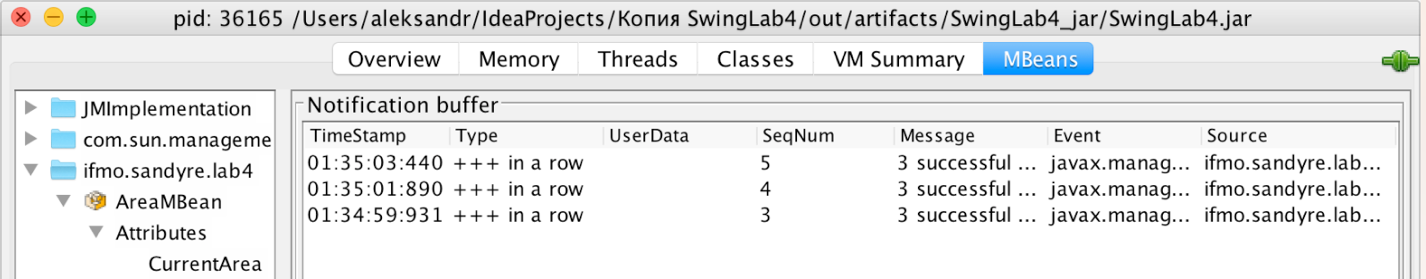
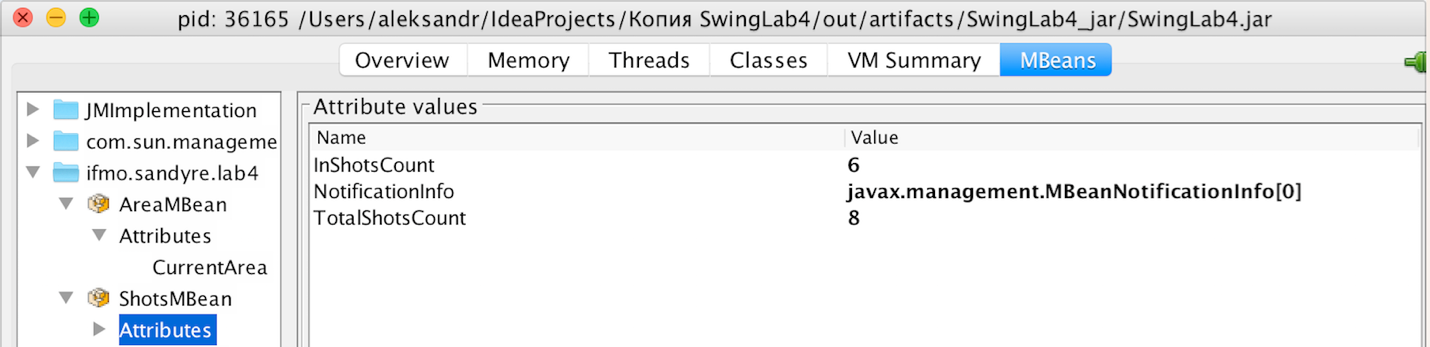
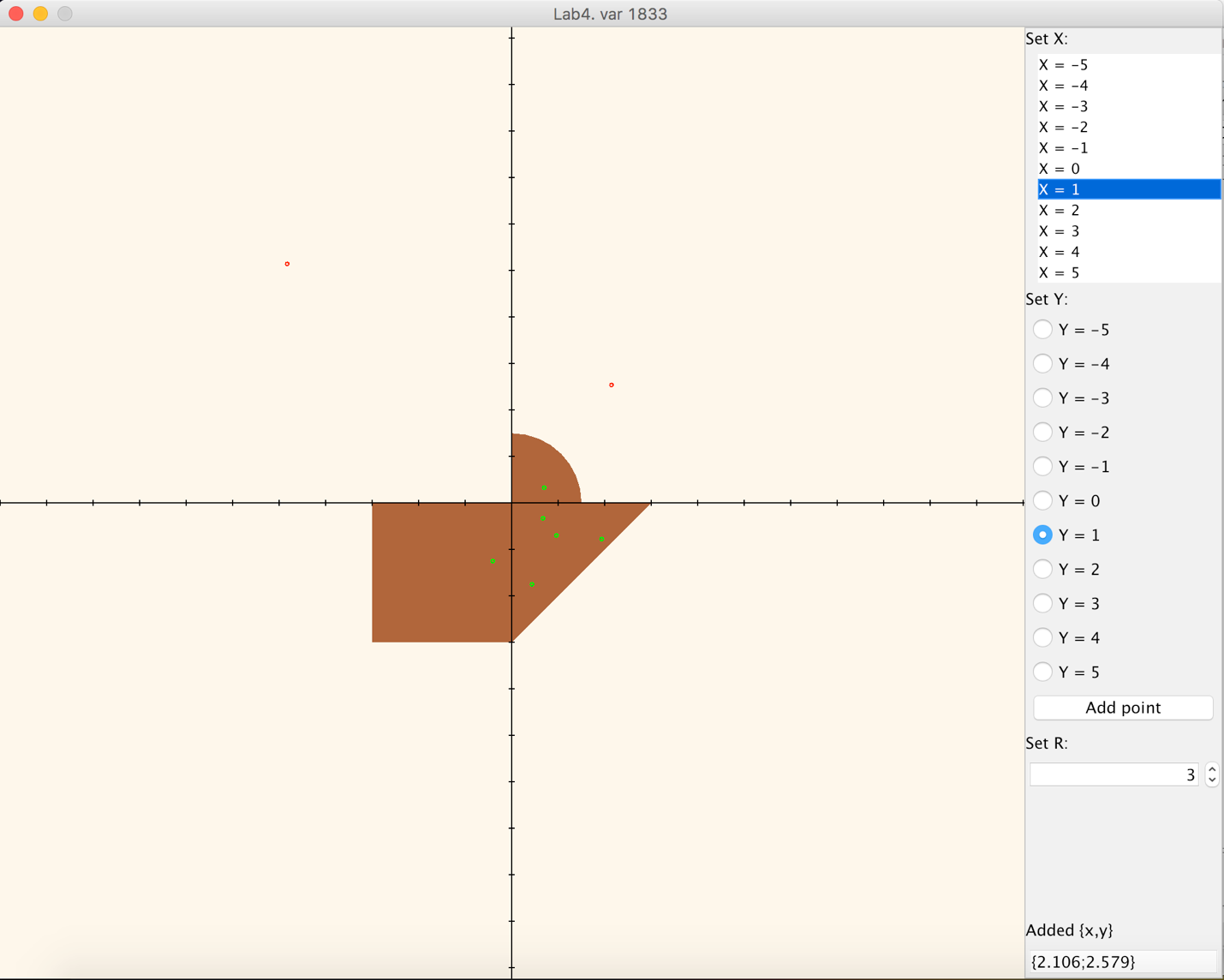
**public class** ShotsController **extends** NotificationBroadcasterSupport  
 **implements** ShotsControllerMBean {  
 **public void** refresh\_and\_add\_shot(**boolean** shot)  
 {  
 **if**(shot)  
 {  
 **successful\_shots\_count**++;  
 **total\_shots\_count**++;  
 **successful\_shots\_in\_row**++;  
 }  
 **else** {  
 **total\_shots\_count**++;  
 **successful\_shots\_in\_row** = 0;  
 }  
  
 **if**(**successful\_shots\_in\_row** == 3)  
 {  
 **successful\_shots\_in\_row** = 0;  
 System.***out***.println(**"oh baby a triple!"**);  
 sendNotification(**new** Notification(**"+++ in a row"**, **this**, **sequence\_number**++, System.*currentTimeMillis*(),  
 **"3 successful shots were made in a row!"**));  
 }  
 }  
  
 **public int** getInShotsCount()  
 {  
 **return successful\_shots\_count**;  
 }  
  
 **public int** getTotalShotsCount()  
 {  
 **return total\_shots\_count**;  
 }  
  
 **private int sequence\_number** = 1;  
 **private int successful\_shots\_in\_row** = 0;  
 **private int total\_shots\_count** = 0;  
 **private int successful\_shots\_count** = 0;  
}

AreaComputer:

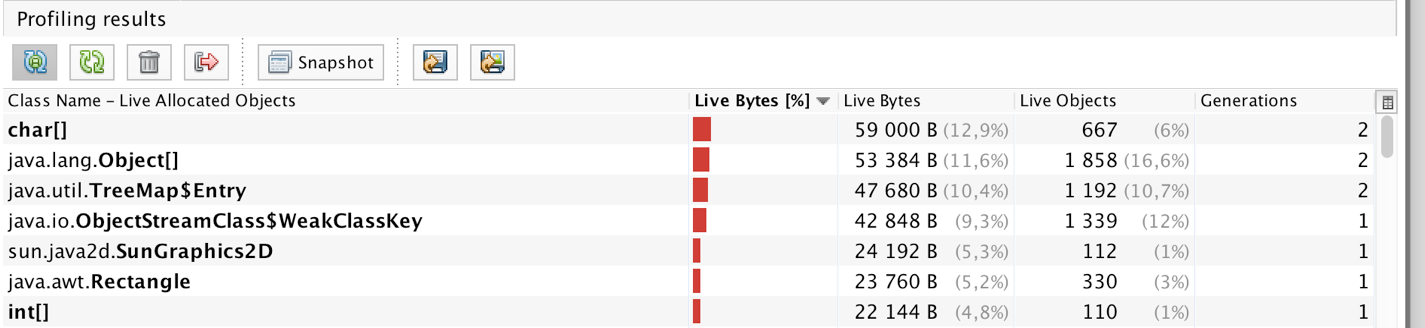
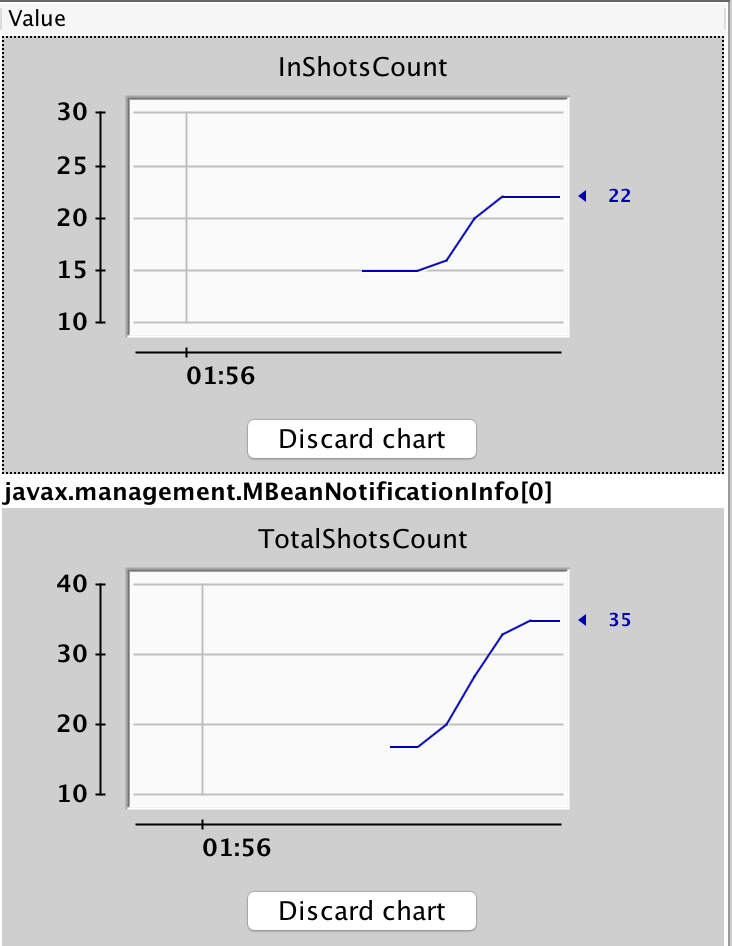
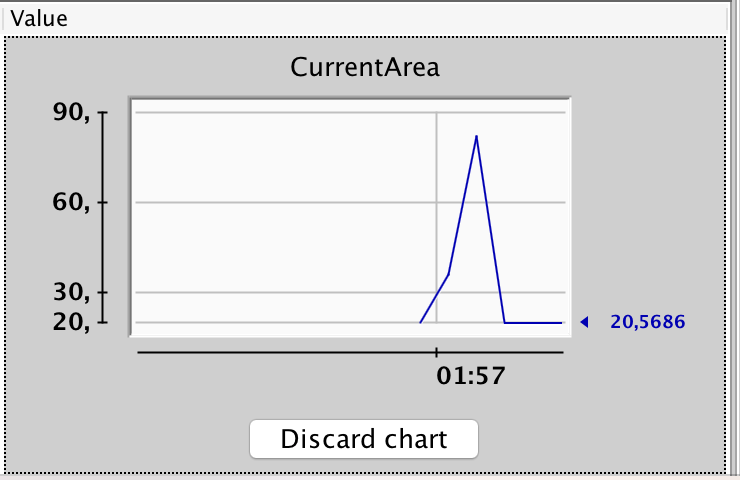
**public interface** AreaComputerMBean {  
 **double** getCurrentArea();  
}

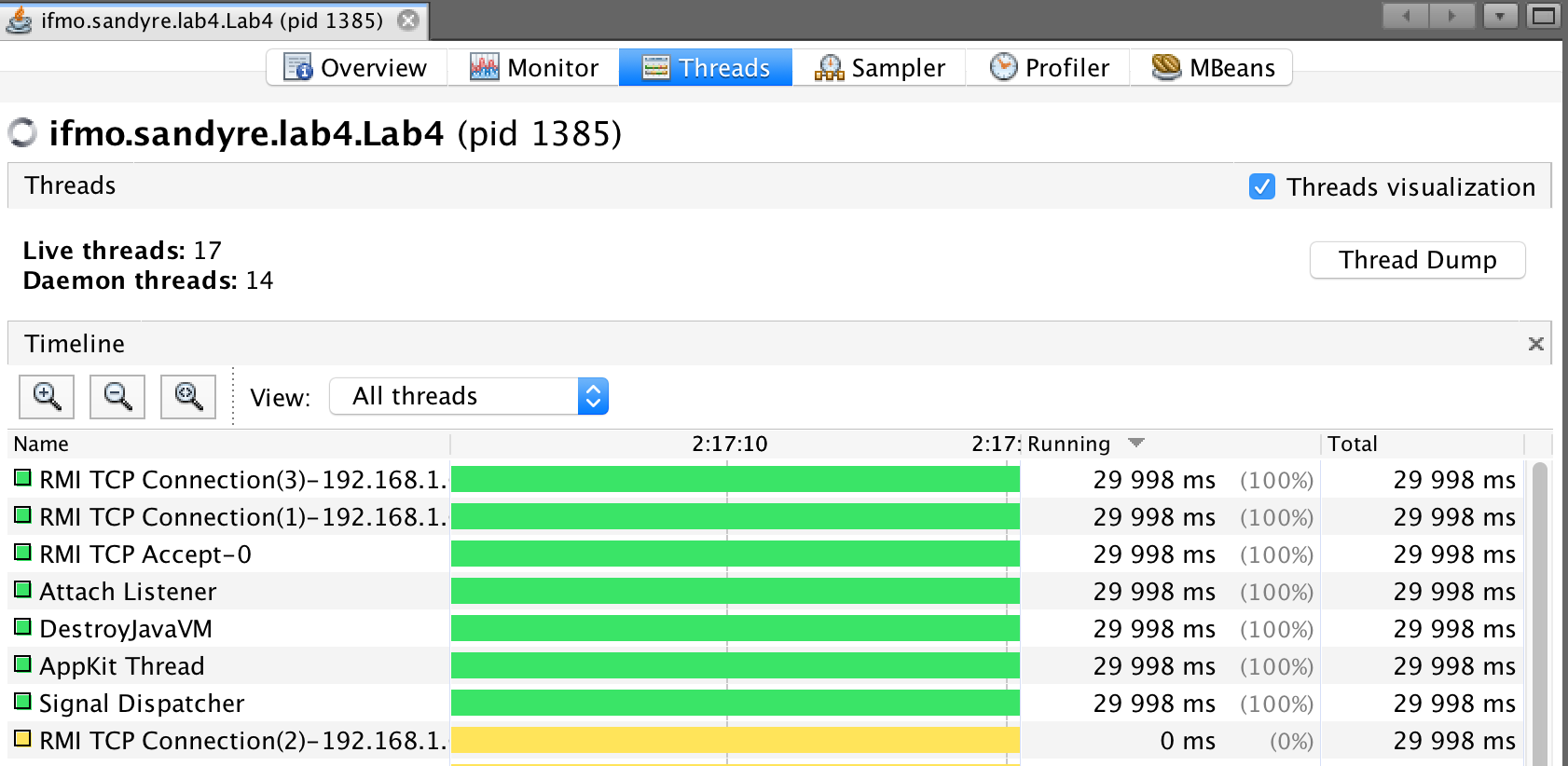
**public class** AreaComputer **implements** AreaComputerMBean {  
 **public** AreaComputer(**double** rad)  
 {  
 **radius** = rad;  
 refreshArea();  
 }  
  
 **public double** getCurrentArea()  
 {  
 **return current\_area**;  
 }  
  
 **public void** refreshArea()  
 {  
 **current\_area** = (**radius** \* **radius**) + (**radius** \* **radius**)/2.0 +  
 (Math.***PI*** \* 0.25 \* **radius** \* **radius**);  
 }  
  
 **public void** setRadius(**double** rad)  
 {  
 **radius** = rad;  
 }  
 **private double radius**;  
 **private double current\_area**;  
}

Мониторинг программы при помощи JConsole



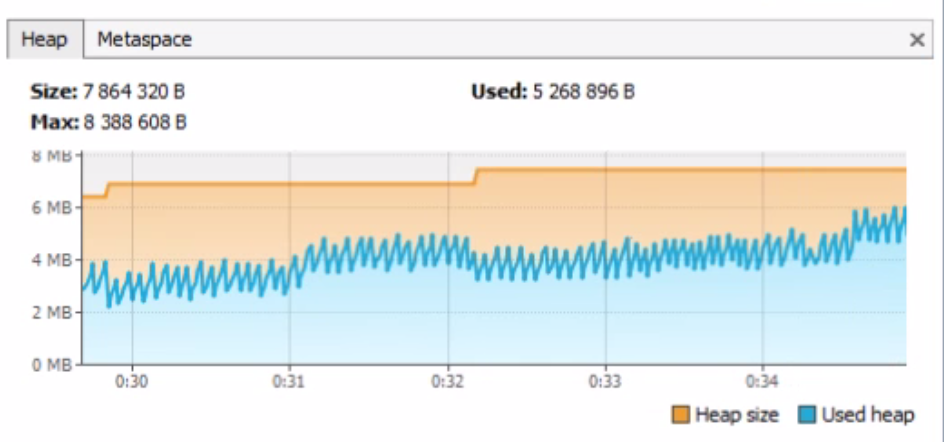
Профилирование программы с помощью VisualVM





Обнаружение и устранение утечки памяти

Запускаем jvisualvm, и приложение с ключами –Xmx8m –Xms5m, чтобы ограничить количество памяти, выделяемое под приложение. По графику видно, что утечка присутствует:



Посмотрев heap dump, можно заметить увеличение размера хеш-мапа. Однако нужно заметить, что в хешмап добавляются элементы с одним и тем же ключем, то есть роста как такового быть не должно.

Утечка заключается в следующей функции:

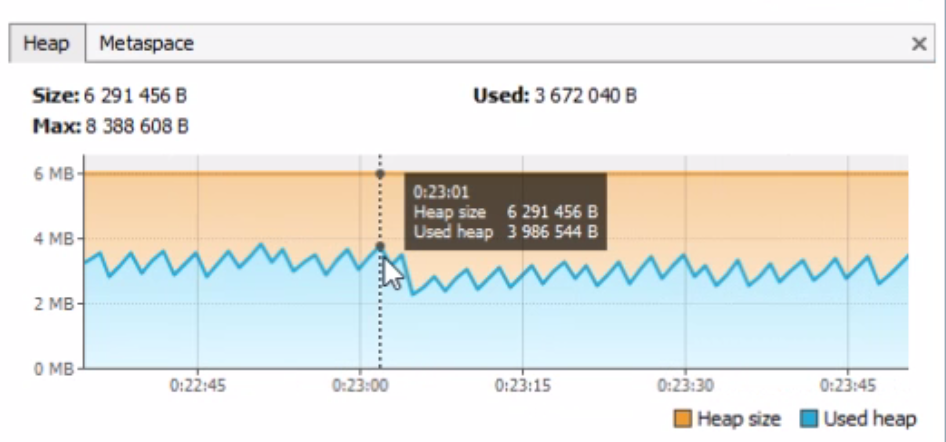
**public byte**[] getValueFromCache(String s) {  
 **try** {  
 java.net.URL url = **new** java.net.URL(s);  
 **if**(!**cache**.containsKey(url)) {  
 **cache**.put(url, **new byte**[786432]);  
 }  
 **return cache**.get(url);  
 } **catch** (Exception e) {  
 System.***out***.println(**"Error: invalid URL!"**);  
 **return null**;  
 }  
}

А конкретно в создании нового экземпляра класса java.net.URL. Экземпляр, который будет создан на основании лишь одного переданного аргумента в конструктор, может отличаться от другого экземпляра, созданного на основании аргумента с идентичным значением. Так как экземпляры будут различны, то и функция проверки наличия ключа в хеш-таблице выдаст false, хотя запрашивается кэш одного и того же сайта.

Исправление утечки:

**public byte**[] getValueFromCache(String s) {  
 **try** {  
 java.net.URL url = **new** java.net.URL(s);Iterator it = **cache**.entrySet().iterator();  
 **while**(it.hasNext())  
 {  
 Map.Entry pair = (Map.Entry)it.next();  
 java.net.URL tmp = (java.net.URL)pair.getKey();  
 **if**(tmp.getAuthority().equals(url.getAuthority()))  
 {  
 **return cache**.get(it);  
 }  
 }  
 **cache**.put(url, **new byte**[786432]);  
 **return cache**.get(url);  
 } **catch** (Exception e) {  
 System.***out***.println(**"Error: invalid URL!"**);  
 **return null**;  
 }  
}

Результат:



Вывод: мониторинг и профилирование осуществляются с целью сбора необходимой разработчику статистики работы программы. Данная статистика может включать и действия пользователя (мониторинг, осуществленный посредством MBean классов в лабораторной работе), так и использование системных ресурсов: использование оперативной памяти, процессорного времени и т.д. Все эти действия осуществляются с целью улучшения программы. Мониторинг позволяет узнать о наиболее частых действиях пользователя, и при необходимости переработать пользовательский интерфейс, попытаться улучшить производительность часто используемых функций и т.д. Профилирование же дает средства для общего изучения производительности системы, поиска участков кода, которые могут оказаться критичными.