

September 2020,  
Programming in Java

# Programming in Java

Trayan Iliev

[tiliev@iproduct.org](mailto:tiliev@iproduct.org)

<http://iproduct.org>

Copyright © 2003-2020 IPT - Intellectual  
Products & Technologies

# About me



## Trayan Iliev

- CEO of IPT – Intellectual Products & Technologies
- Oracle® certified programmer 15+ Y
- end-to-end reactive fullstack apps with Java, ES6/7, TypeScript, Angular, React and Vue.js
- 12+ years IT trainer
- Voxxed Days, jPrime, jProfessionals, BGOUG, BGJUG, DEV.BG speaker
- Organizer RoboLearn hackathons and IoT enthusiast

# What Will You Learn?

1. Java classes and objects – 4h
2. Core data structures & generics – 3h
3. OOP principles – Abstraction, Encapsulation, Inheritance, Understanding Polymorphism, SOLID principles – 4h
4. Exception handling – 2h
5. Core Java classes – String, StringBuilder, functional programming, Stream API, Java Date/Time API, NIO2 API, JDBC – 12h
6. Creating Java Application – 4h
7. Debugging – 1h
8. Web basics – 5h

# Course Schedule

- ❖ Block 1: 9:00 – 11:00
  - ❖ Pause: 11:00 – 11:10
  - ❖ Block 2: 11:10 – 12:40
  - ❖ Lunch: 12:40 – 13:10
  - ❖ Block 3: 13:10 – 14:40
-

# Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

<https://github.com/iproduct/course-java-fd>



# Agenda for This Session

- **Java Class structure** – package, imports, fields, methods, access modifiers;
- **Creating objects** – constructors, order of initialization, static members, keyword this, constructors overloading;
- **Working with methods** – designing methods, arguments and return values, overloading, static methods, access modifiers;
- **Define the scope of variables** – class(static), local, instance variables;
- **Apply encapsulation** principles to a class;
- **Understand objects equality** – the difference between “==” and equals();
- **Wrapper Classes**;
- **Distinguish between Object reference and primitive variables**, type casting; Methods reference and primitive arguments;
- **Enumerations**;
- **Object lifecycle** – destroying objects, garbage collection – finalize();

# Key Features of Java Language

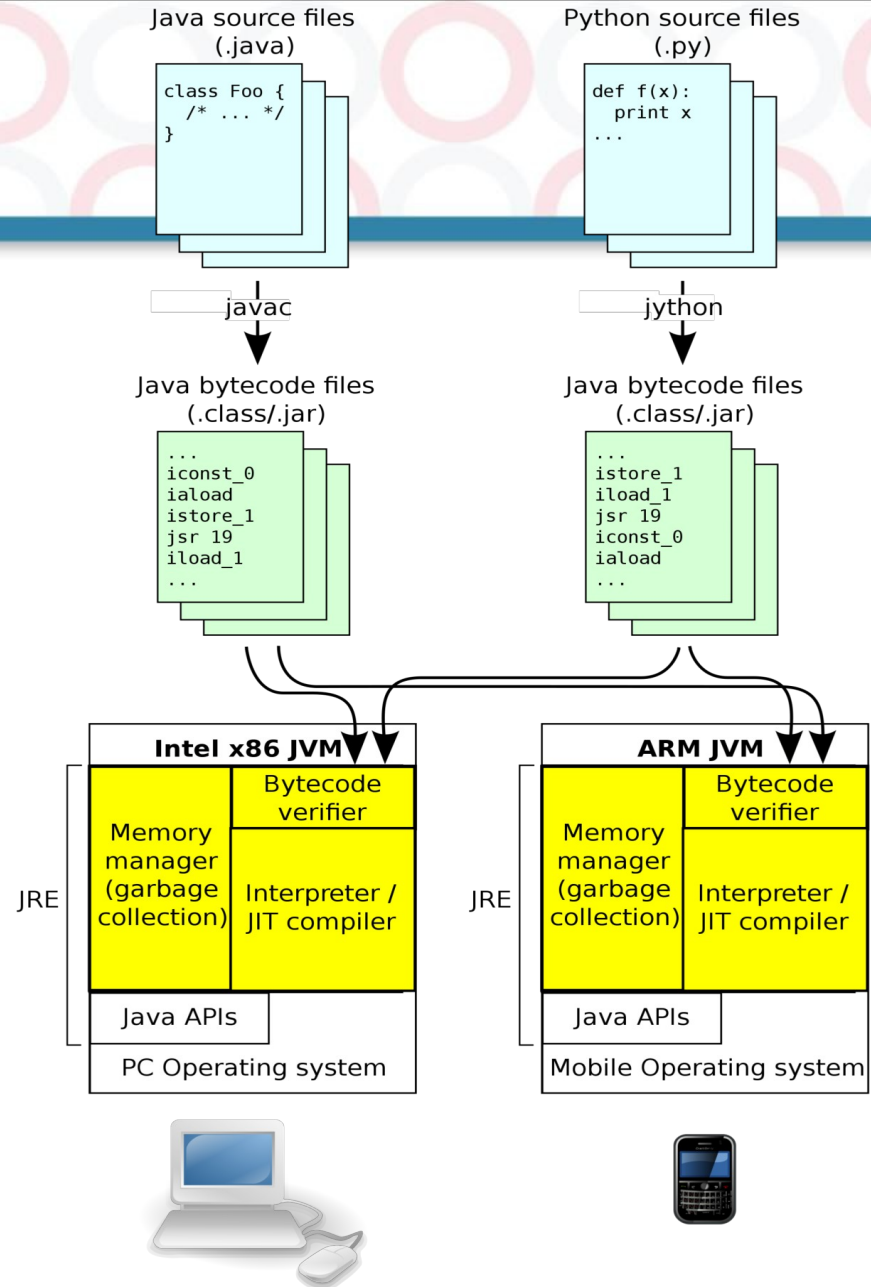
- **Single base hierarchy** - inheritance from only one parent class, with the possibility of implementation of multiple interfaces
- **Garbage Collector** – portability and platform independence, fewer errors
- **Secure Code** – separation of business logic from the error handling and exceptions
- **Multithreading** - easy realization of parallel processing
- **Persistence** – Java Database Connectivity (JDBC) and Java Persistence API (JPA)

# Integrated Development Environments for Java Applications

- Java™ development environment types:
- JavaSE, JavaEE, JavaME, JavaFX
- JavaSE: Java Development Kit (JDK) and Java Runtime Environment (JRE)
- Java™ compiler - javac
- Java Virtual Machine (JVM) - java
- Source code → Byte code
- Installing JDK 8+
- Compile and run programs from the command line
- IDEs: IntelliJ IDEA, Eclipse



# Java Virtual Machine (JVM)



# Java Application Stack

**Java™ Custom Application** – Level & patterns of garbage production, Concurrency, IO/Net, Algorithms & Data structures, API & Frameworks

**Application Server** – Web Container, EJB Container, Distributed Transactions Dependency Injection, Persistence - Connection Pooling, Non-blocking IO

**Java™ Virtual Machine (JVM)** – Gartbage Collection, Threads & Concurrency, NIO

**Operating System** – Virtual Memory, Paging, OS Processes and IO/Net libraries

**Hardware Platform** – CPU, Memory, IO, Network

Processing Node 1

Processing Node2

...

Processing Node N

Level of Optimization ↓

# Classes, Objects and References

- **Class** - set of objects that share a common structure, behaviour and possible links to objects of other classes = **objects type**
  - ✓ **structure** = attributes, properties, member variables
  - ✓ **behaviour** = methods, operations, member functions, messages
  - ✓ **relations** between classes: **association, inheritance, aggregation, composition** – modeled as attributes (**references** to objects from the connected class)
- **Objects** are instances of the class, which is their addition:
  - ✓ own state
  - ✓ unique identifier = reference pointing towards object

# Object (Reference) Data Types

- Creating a class (a new data type)

```
class MyClass { /* attributes and methods of the class */ }
```

- Create an object (instance) from the class MyClass :

```
MyClass myObject = new MyClass();
```

- Declaration and initialization of attributes:

```
class Person {  
    String name = "Anonymous";  
    int age;  
}
```

- Access to attribute: 

```
Person p1 = new Person();  
p1.name = "Ivan Petrov";    p1.age = 28;
```

# Creating Objects

- Class **String** – modeling string of characters:

- **declaration**:

```
String s;
```

- **initialization** (on separate line):

```
s = new String("Hello Java World");
```

- **declaration + initialization**:

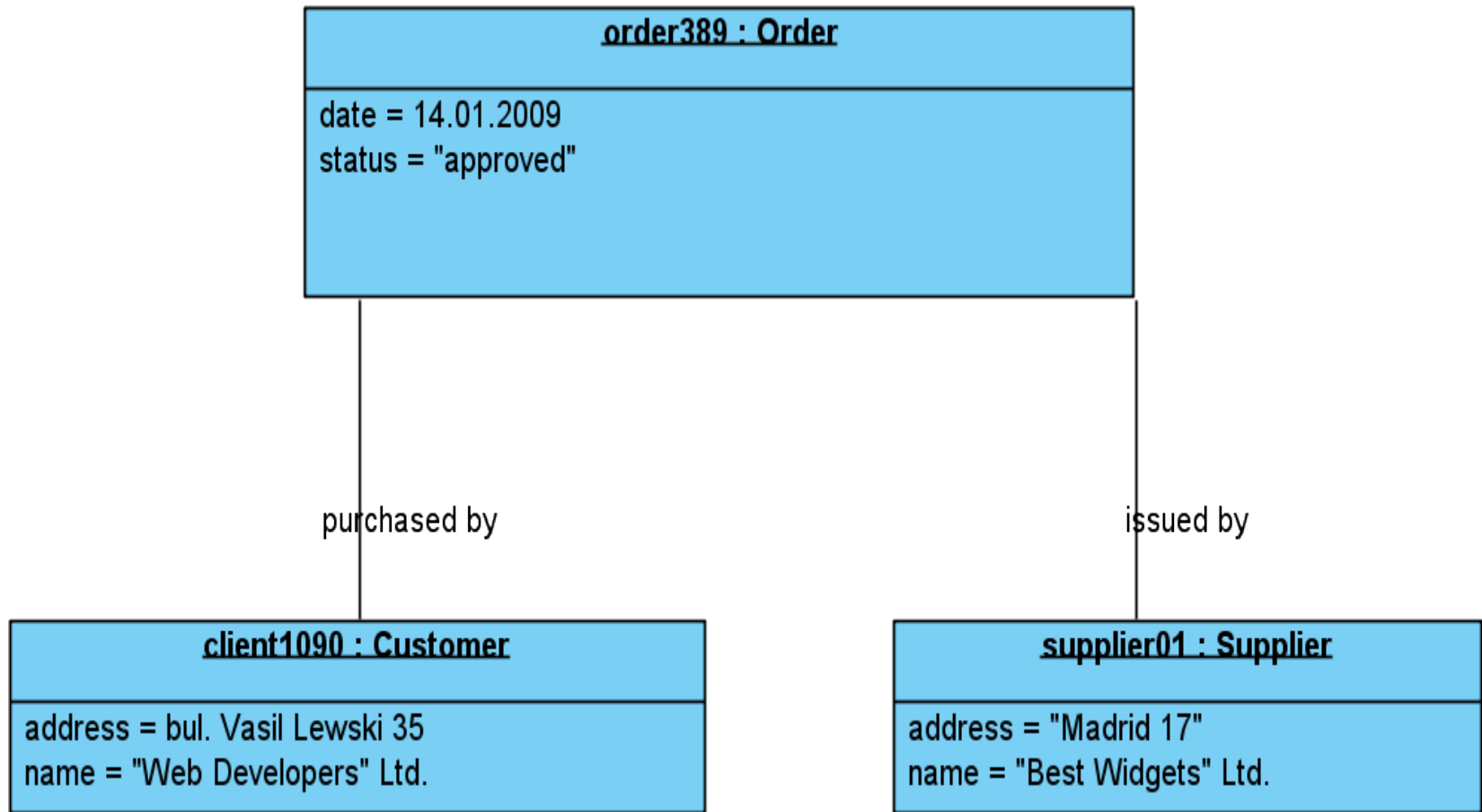
```
String s = new String("Hello Java World");
```

- **declaration + initialization** (shorter form, applies only to the class String):

```
String s = "Hello Java World";
```



# Object Diagram



# Packages and Access Specifiers

- ❖ Packages and directories
- ❖ Importing packages – import
- ❖ Access specifiers
  - **public**
  - **private**
  - **protected**
  - **Friendly access** – by default within the package

# Primitive and Object Data Types

- **Primitive** data types, **object wrapper** types and default values for attributes of primitive type

– boolean	-->	Boolean	false
– char	-->	Character	'\u0000'
– byte	-->	Byte	(byte) 0
– short	-->	Short	(short) 0
– int	-->	Integer	0
– long	-->	Long	0L
– float	-->	Float	0.0F
– double	-->	Double	0.0D
– void	-->	Void	

❖ **BigInteger** and **BigDecimal** - higher-precision numbers

# Primitive Type Literals

- in decimal notation:  
    int: 145, 2147483647, -2147483648  
    long: 145L, -1L, 9223372036854775807L  
    float: 145F, -1f, 42E-12F, 42e12f  
    double: 145D, -1d, 42E-12D, 42e12d
- in hexadecimal notation: 0x7ff, 0x7FF, 0X7ff, 0X7FF
- in octal notation: 0177
- in binary notation: 0b11100101, 0B11100101

# Object (Reference) Data Types

- Initialization with default values
- Value of uninitialized reference = **null**
- Declaring class methods

```
class Person {
```

```
    String name;
```

```
    int age;
```

```
    String changeNameAndAge (String aName, int anAge) {
```

```
        name = aName;
```

```
        age = anAge;
```

```
        return "Name: " + name + "Age: " + age;
```

```
    }
```

```
}
```

Method Name

Arguments

Return Type

Method Body

Returning Value



# Object Constructors in Java

- Initialization of objects with constructors
- **Overloading** of constructors and other methods
- Default constructors
- Reference to the current object – **this**

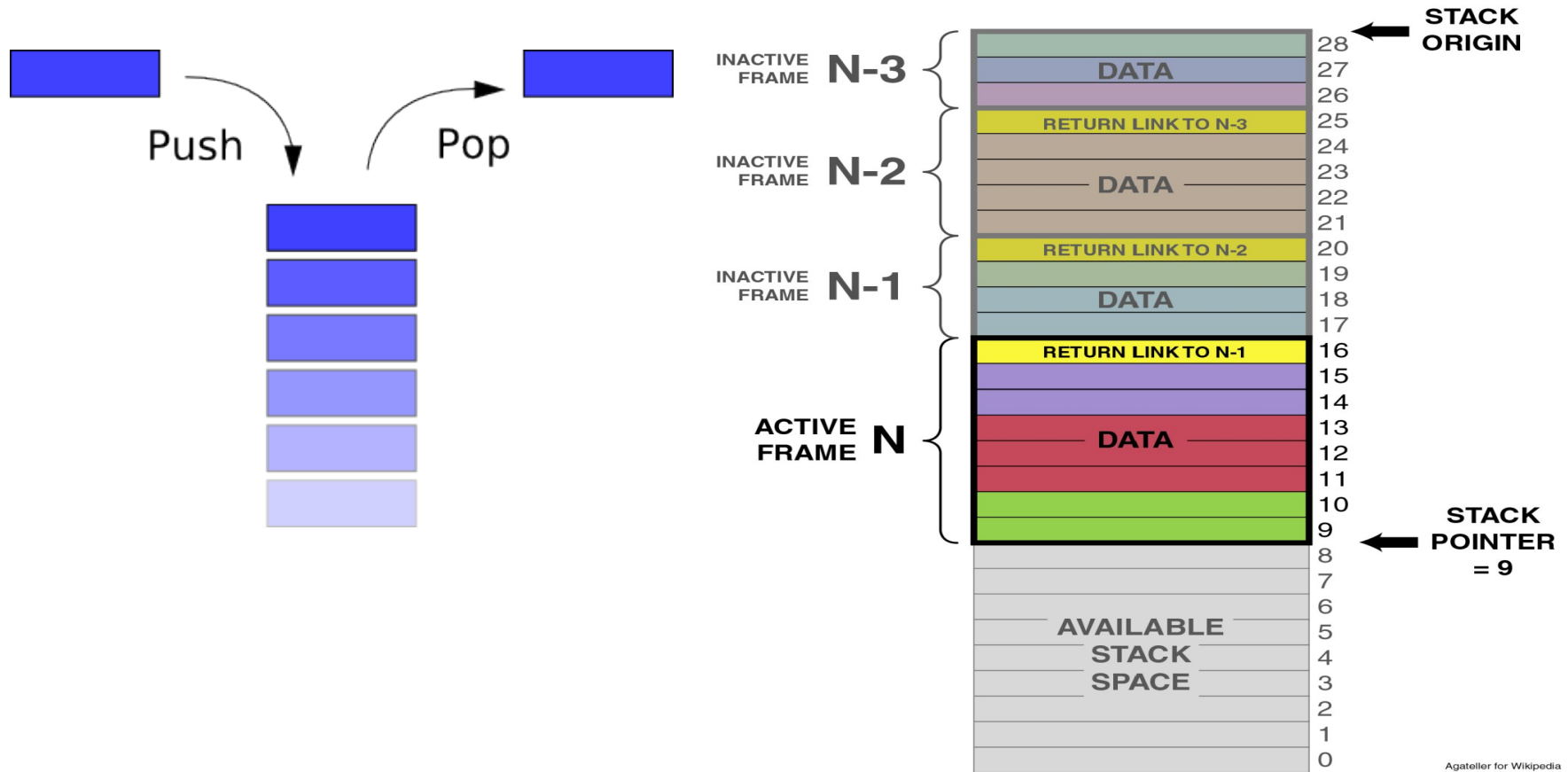
# Objects Initialization. Array initialization

- Initialization in declaration
- Initialization in constructor
- „Lazy“ initialization
- Initialization of static class members
- One-dimensional and multi-dimensional arrays
- Array initialization

# Memory Types

- **Register memory** - CPU registers, fast, small numbers stored operand instructions just before treatment
- **Program Stack** = Last In, First Out (LIFO) – Keep primitive data types and references to objects during program execution
- **Dynamically allocated memory – Heap** – can store different sized objects for different periods of time, can create new objects dynamically and to be released – Garbage Collector
  - Young generation – objects that exist for short period
  - Old generation – objects that exist longer
  - Permanent Generation = class definitions. **Java 8+ Metaspace**
- **Constant storage, non-RAM storage (external memory)**

# Program Stack



Agateller for Wikipedia  
Public Domain 2006

c:\CourseAdvancedJavaVerint\Temp&gt;jstack 1612

2015-07-16 15:52:18

Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.45-b02 mixed mode):

```
"DestroyJavaVM" #21 prio=5 os_prio=0 tid=0x0000000024b8000 nid=0x1f04 waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

"Thread-9" #20 prio=5 os_prio=0 tid=0x00000000bea7000 nid=0x2348 waiting for monitor entry [0x00000000d14f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-8" #19 prio=5 os_prio=0 tid=0x00000000bea5800 nid=0x6ac waiting for monitor entry [0x00000000ca2e000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-7" #18 prio=5 os_prio=0 tid=0x00000000bea5000 nid=0x1ffc waiting for monitor entry [0x00000000cfcf000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-6" #17 prio=5 os_prio=0 tid=0x00000000bea2000 nid=0x40c waiting for monitor entry [0x00000000cd5f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-5" #16 prio=5 os_prio=0 tid=0x00000000bea0800 nid=0x1708 waiting for monitor entry [0x00000000ceae000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

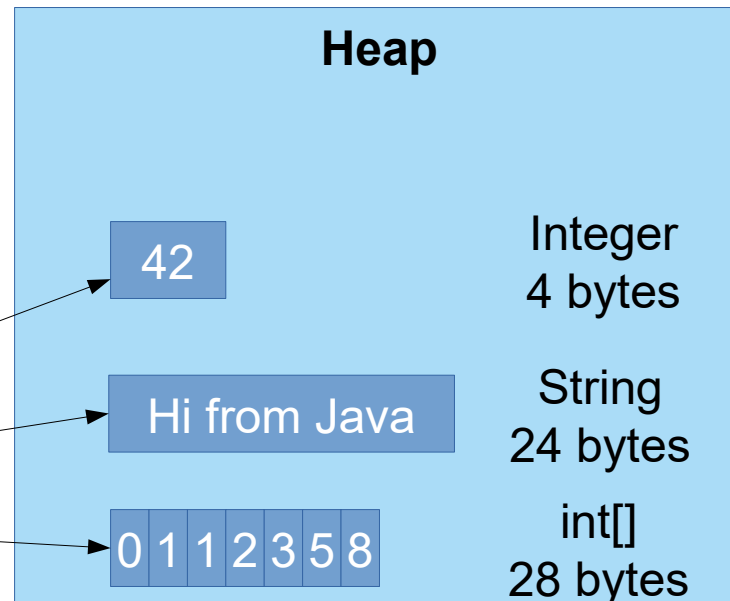
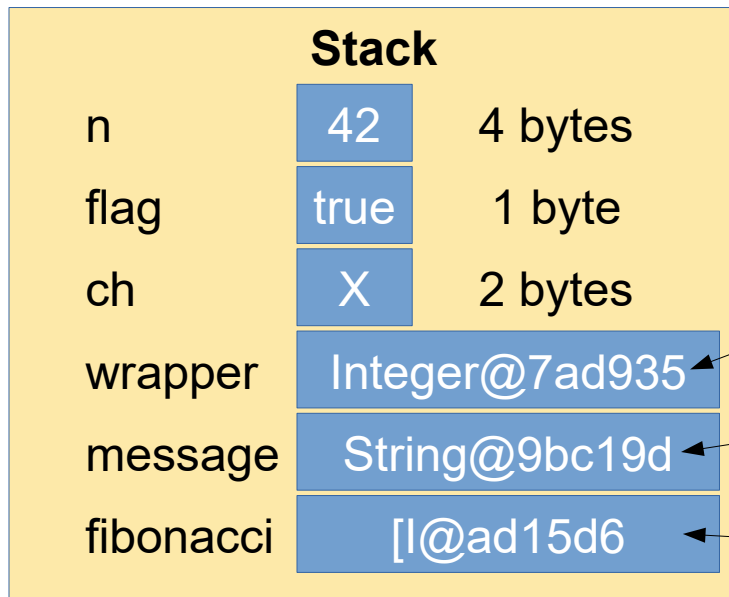
"Thread-4" #15 prio=5 os_prio=0 tid=0x00000000be9d000 nid=0xc0c waiting for monitor entry [0x00000000c7df000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at simpletest.TwoThreadsSynchronizedCounter.lambda$0(TwoThreadsSynchronizedCounter.java:14)
    - waiting to lock <0x00000000d5e660a0> (a java.lang.Object)
    at simpletest.TwoThreadsSynchronizedCounter$$Lambda$1/424058530.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:745)

"Thread-3" #14 prio=5 os_prio=0 tid=0x00000000be9c800 nid=0x2394 waiting for monitor entry [0x00000000cc2f000]
  java.lang.Thread.State: BLOCKED (on object monitor)
```



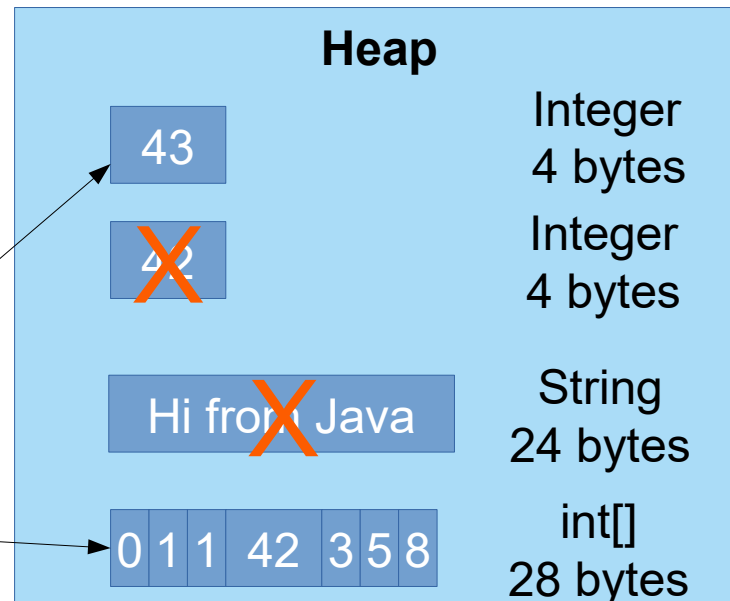
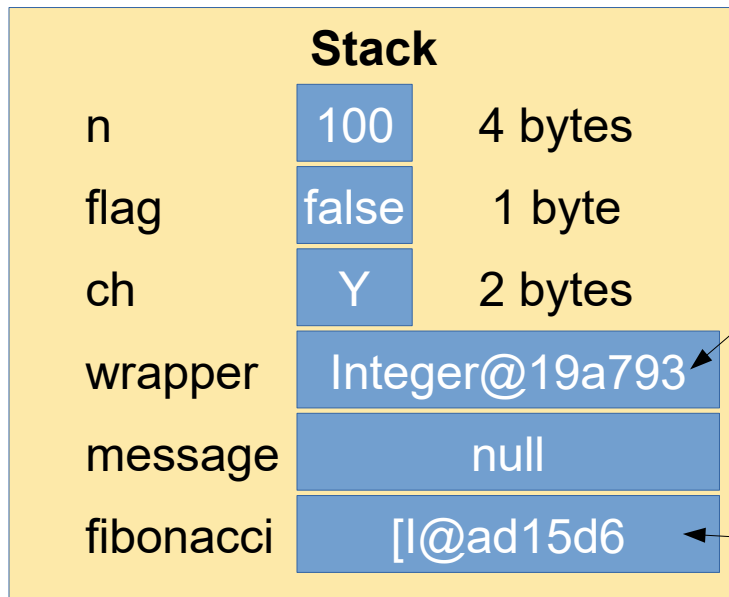
# Stack and Heap (Quick Review)

```
int n = 42;
boolean flag = true;
char ch = 'X';
Integer wrapper = n;
String message = "Hi from Java!";
int[] fibonacci = { 0, 1, 1, 2, 3, 5, 8 };
```



# Stack and Heap (Quick Review)

```
n = 100;  
flag = !flag;  
h = ++ch;  
wrapper = ++wrapper;  
message = null;  
fibonacci[3] = 42;
```



# Variable Scopes

```
public class VarScopes {  
    static int s1 = 25;  
    int i1 = 350;  
    public static void main(String[] args) {  
        if(s1 > 10){  
            int a = 42;  
            // Only a available  
            {  
                int b = 108; // Both a & b are available  
            }  
            // Only a available, b is out of scope  
        }  
        // a & b are out of scope  
    }  
}
```

# Operators in Java - I

- Assignment operator
- Mathematical operators
- Relational operators
- Logical operators
- Bitwise operators
- String operators
- Operators for type conversion
- Priorities of operators

# Operators in Java - II

- Each operator has priority and associativity - for example,  $+$  and  $-$  have a lower priority from  $*$  and  $/$
- The priority can be set clearly using brackets ( and ) - for example  $(y - 1) / (2 + x)$
- According associativity operators are left-associative, right-associative and non-associative: For example:  
 $x + y + z \Rightarrow (x + y) + z$ , because the operator  $+$  is left-associative
- if it was right associative, the result would be  
 $x + (y + z)$



# Operators in Java - III

- Assignment operator: **=**
  - is not symmetrical – i.e. **x = 42** is OK, **42 = x** is NOT
  - to the left always stands a variable of a certain type, and to the right an expression from the same type or type, which can be automatically converted to present
- Mathematical operators:
  - with one argument (unary): **-, ++, --**
  - with two arguments (binary): **+, -, \*, /, %** (remainder)
- Combined: **+=, -=, \*=, /=, %=**

For example: **a += 2**  $\Leftrightarrow$  **a = a + 2**

# Send Arguments by Reference or by Value

- Formal and actual arguments - Example:

Static method - no **this**

Formal Argument  
- copies the actual value

```
public static void incrementAgeBy10(Person p){  
    p.age = p.age + 10;  
}
```

```
Person p2 = new Person(23434345435L, "Petar Georgiev",  
    "Plovdiv", 39);
```

```
incrementAgeBy10(p2);
```

Actual Argument

```
System.out.println(p2);
```

# Send Arguments by Reference and Value

- **Case A:** When the argument is a primitive type, the formal argument copies the actual value
- **Case B:** When the argument is a **object type**, the formal argument **copies reference** to the actual value
- **Cases A & B:** Changes in the copy (formal argument) **does not reflect** the actual argument
- However, if formal and actual argument point to the same object (**Case B**) – then **changes in properties (attribute values) of this object are available from the calling method** – i.e. we can return value from this argument

# Operators in Java - IV

- Relational operators (comparison): **==, !=, <=, >=**
- Logical operators: **&& (AND), || (OR)** and **! (NOT)**  
the expression is calculated from left to right **only when it's necessary** for determining the final outcome
- Bitwise operators: **& (AND), | (OR)** and **~ (NOT), ^ (XOR),  
&=, |=, ^=**
- Bitwise shift: **<<, >>** (preserves character), **>>>** (always inserts zeros left – does not preserve character), **<<=, >>=, >>>=**

# Operators in Java - V

- Triple **if-then-else** operator:

**<boolean-expr> ? <then-value> : <else-value>**

- String concatenation operator: **+**

- Operators for type conversion (type casting):

**(byte), (short), (char), (int), (long), (float) ...**

- Priorities of operators:

**unary > binary arithmetical > relational > logical > three-argumentative operator if-then-else > operators to assign a value**

# Controlling Program Flow - I

- Conditional operator - **if-else**
- Returning Value – **return**
- Operators organizing cycle - **while, do while, for, break, continue**
- Operator to select one from many options - **switch**



# Controlling Program Flow - II

- Conditional operator **if-else**:

**if(<boolean-expr>)**  
    **<then-statement>**

or

**if(<boolean-expr>)**  
    **<then-statement>**  
**else**  
    **<else-statement>**

# Controlling Program Flow - III

- Returning value to exit the method: **return;** or **return <value>;**
- Operator to organize cycle **while**:  
**while(<boolean-expr>)**  
**<body-statement>**
- Operator to organize cycle **do-while**:  
**do <body-statement>**  
**while(<boolean-expr>);**

# Controlling Program Flow - IV

- Operator to organize cycle **for**:

**for(<initialization>; <boolean-expr>; <step>)  
<body-statement>**

- Operator to organize cycle **foreach**:

**for(<value-type> x : <collection-of-values>)  
<body-statement-using-x>**

Ex.: **for(Point p : pointsArray)**

**System.out.println("(" + p.x + ", " + p.y + ");");**

# Controlling Program Flow - V

- Operators to exit block (cycle) **break** and to exit iteration cycle **continue**:

```
<loop-iteration> {
```

```
//do some work
```

```
continue; // goes directly to next loop iteration
```

```
//do more work
```

```
    break; // leaves the loop
```

```
    //do more work
```

```
}
```

# Controlling Program Flow - VI

- Use of labels with **break** and **continue**:

**outer\_label:**

```
<outer-loop> {  
    <inner-loop> {  
        //do some work  
        continue; // continues inner-loop  
        //do more work  
        break outer_label; // breaks outer-loop  
        //do more work  
        continue outer_label; // continues outer-loop  
    }  
}
```

# Controlling Program Flow - VII

❖ Selecting one of several options **switch**:

```
switch(<selector-expr>) {  
  case <value1> : <statement1>; break;  
  case <value2> : <statement2>; break;  
  case <value3> : <statement3>; break;  
  case <value4> : <statement4>; break;  
  // more cases here ...  
  default: <default-statement>;  
}
```



# Enumeration Types

```
public class MyEnumeration {  
    public enum InvoiceType { SIMPLE, VAT }  
    public static void main(String[] args) {  
        for(InvoiceType it : InvoiceType.values())  
            System.out.println(it);  
    }  
}
```

Результат: *SIMPLE*  
*VAT*

# Garbage Collection – Main Concepts

- Garbage collection and finalization – method **finalize()**
- Client and Server VMs (~~≠ JIT Compilers & Defaults~~), ~~x86, x64~~
- Generational Garbage Collection – **Young, Old & Permanent** (in Java 8 → **Metaspace**) – Weak generational hypothesis:
  - Most of the objects become unreachable soon;
  - Small number of references exist from old to young objects.

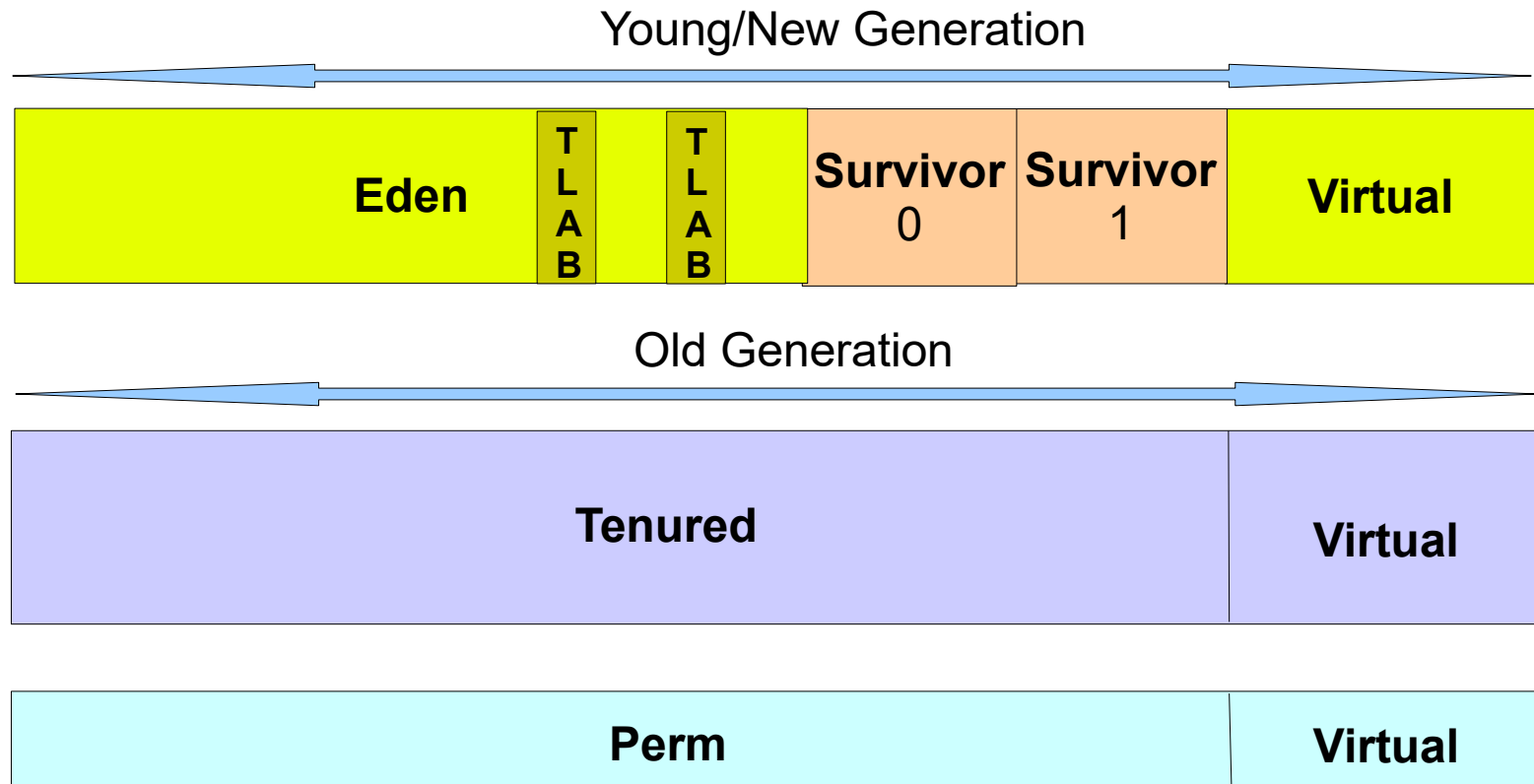
- Tuning for **Higher Throughput**:

```
java -d64 -server -XX:+AggressiveOpts -XX:+UseLargePages -Xmn10g -Xms26g -Xmx26g
```

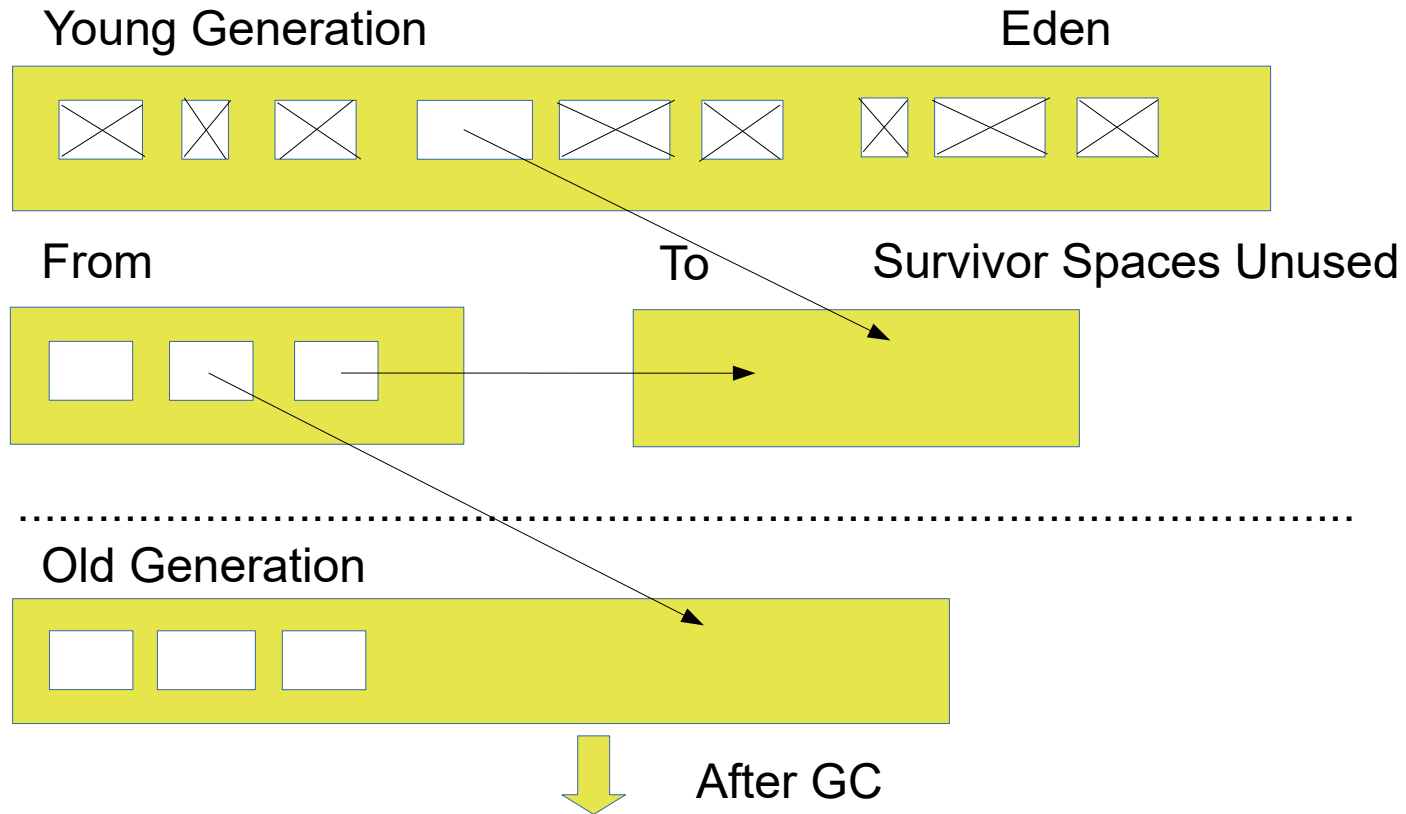
- Tuning for **Lower Latency**

```
java -d64 -XX:+UseG1GC -Xms26g Xmx26g -XX:MaxGCPauseMillis=500 -XX:+PrintGCTimeStamp
```

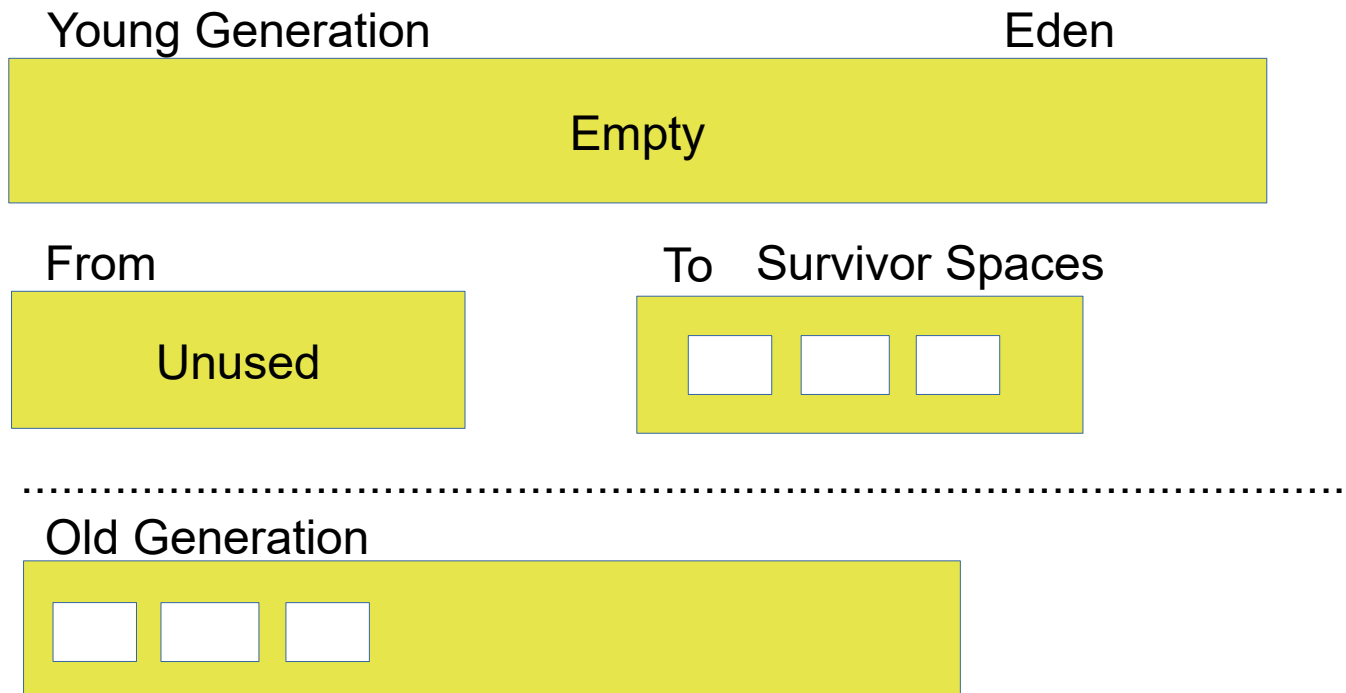
# Garbage Collection – Main Concepts



# Before GC



# After GC



# Garbage Collection – Basic Settings

- Xms** – Heap area size when starting JVM
- Xmx** – Maximum heap area size
- Xmn, -XX:NewSize** – размер на young generation (nursery)
- XX:MinHeapFreeRatio=<N>** –  
**XX:MaxHeapFreeRatio=<N>**
- XX:NewRatio** – Ratio of New area and Old area
- XX:NewSize**    **-XX:MaxNewSize** – New area size  $\leq$  Max
- XX:SurvivorRatio** – Ratio of Eden area and Survivor area
- XX:+PrintTenuringDistribution** – treshhold and ages of New generation
- XX:+PrintGCDetails**
- XX:+PrintGCTimeStamps**



# GC Strategies and Settings

Serial GC **-XX:+UseSerialGC**

Parallel GC **-XX:+UseParallelGC**

**-XX:ParallelGCThreads=<N>**

Parallel Compacting GC **-XX:+UseParallelOldGC**

Conc. Mark Sweep CMS GC **-XX:+UseConcMarkSweepGC**

**-XX:+UseParNewGC**

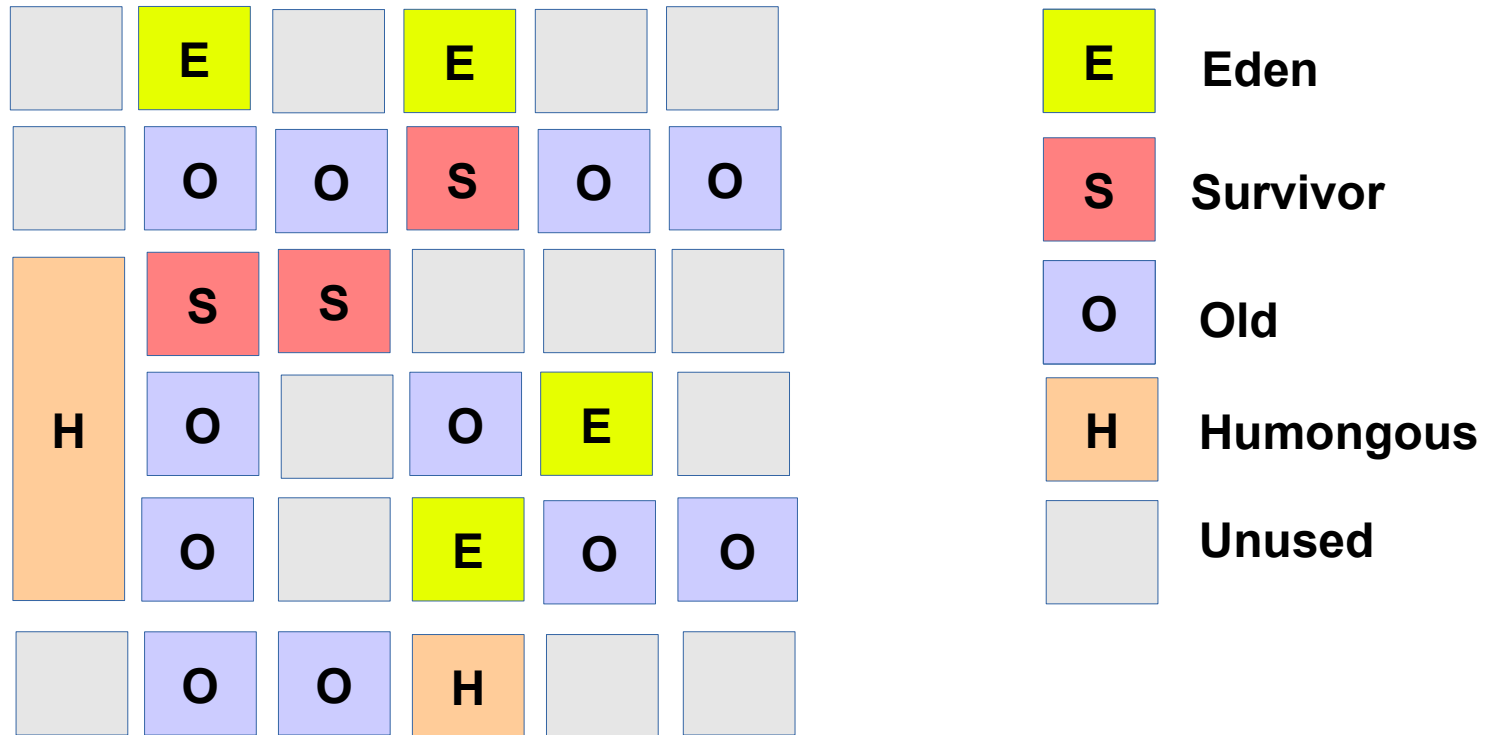
**-XX:+CMSParallelRemarkEnabled**

**-XX:CMSInitiatingOccupancyFraction=<N>**

**-XX:+UseCMSInitiatingOccupancyOnly**

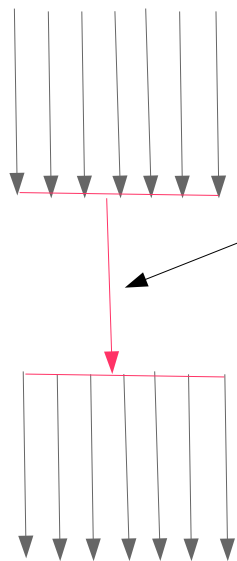
G1 **-XX:+UseG1GC**

# Garbage First G1 Partially Concurrent Collector



# CMS GC (-XX:+UseConcMarkSweepGC)

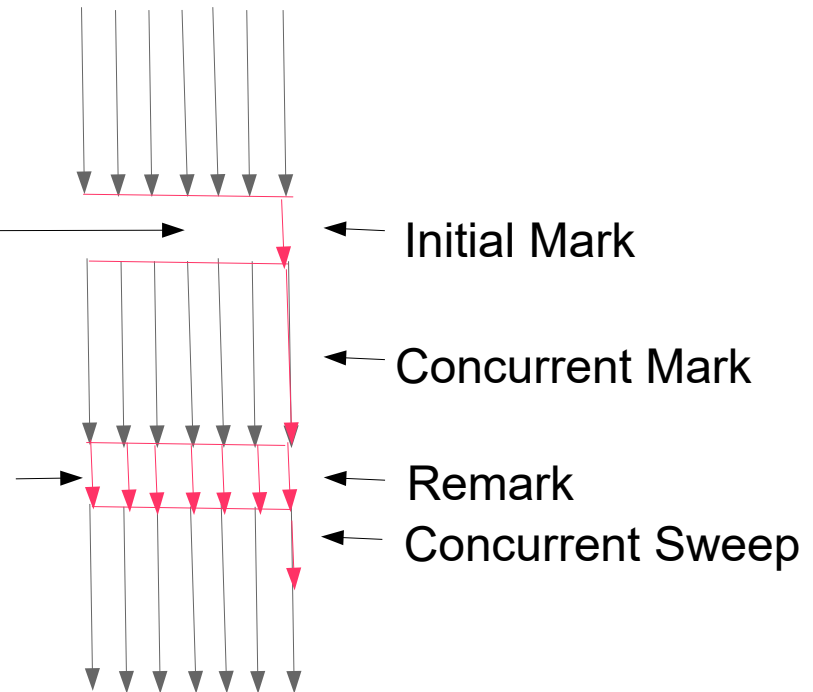
Serial Mark-Sweep-Compact Collector



Stop-the-world pause

Stop-the-world pause

Concurrent Mark-Sweep Collector



Initial Mark

Concurrent Mark

Remark

Concurrent Sweep

# Profiling Recommendations: GC

- **Garbage Collection** – be sure to minimize the GC interference by calling **System.gc()** several times before benchmark start. Call **System.runFinalization()** also. GC activity can be monitored using **-verbose:gc** JVM command. Another way to minimize GC interference is to use serial garbage collector using **-XX:+UseSerialGC** and same value for **-Xmx** and **-Xms**, as well as explicitly setting **-Xnm** flags.
- Use more precise **System.nanoTime()**, but be aware that the time can be reported with varying degree of accuracy in different JVM implementations.

# Java Command Line Monitoring/Tuning Tools - I

**jps** – reports the local VM identifier (**lvmid** - typically the process identifier - **PID** for the JVM process), for each instrumented JVM found on the target system.

**jcmd** – reports class, thread and VM information for a java process: **jcmd <PID> <command> <optional arguments>**

**jinfo** – provides information about current system properties of the JVM and for some properties allows to be set dynamically:

**jinfo -sysprops <PID>**

**jinfo -flags <PID>**

**jinfo -flag PrintGCDetails <PID>**

**jinfo -flag -PrintGCDetails <PID>** - sets **-XX:-PrintGCDetails**

# Java Command Line Monitoring/Tuning Tools -II

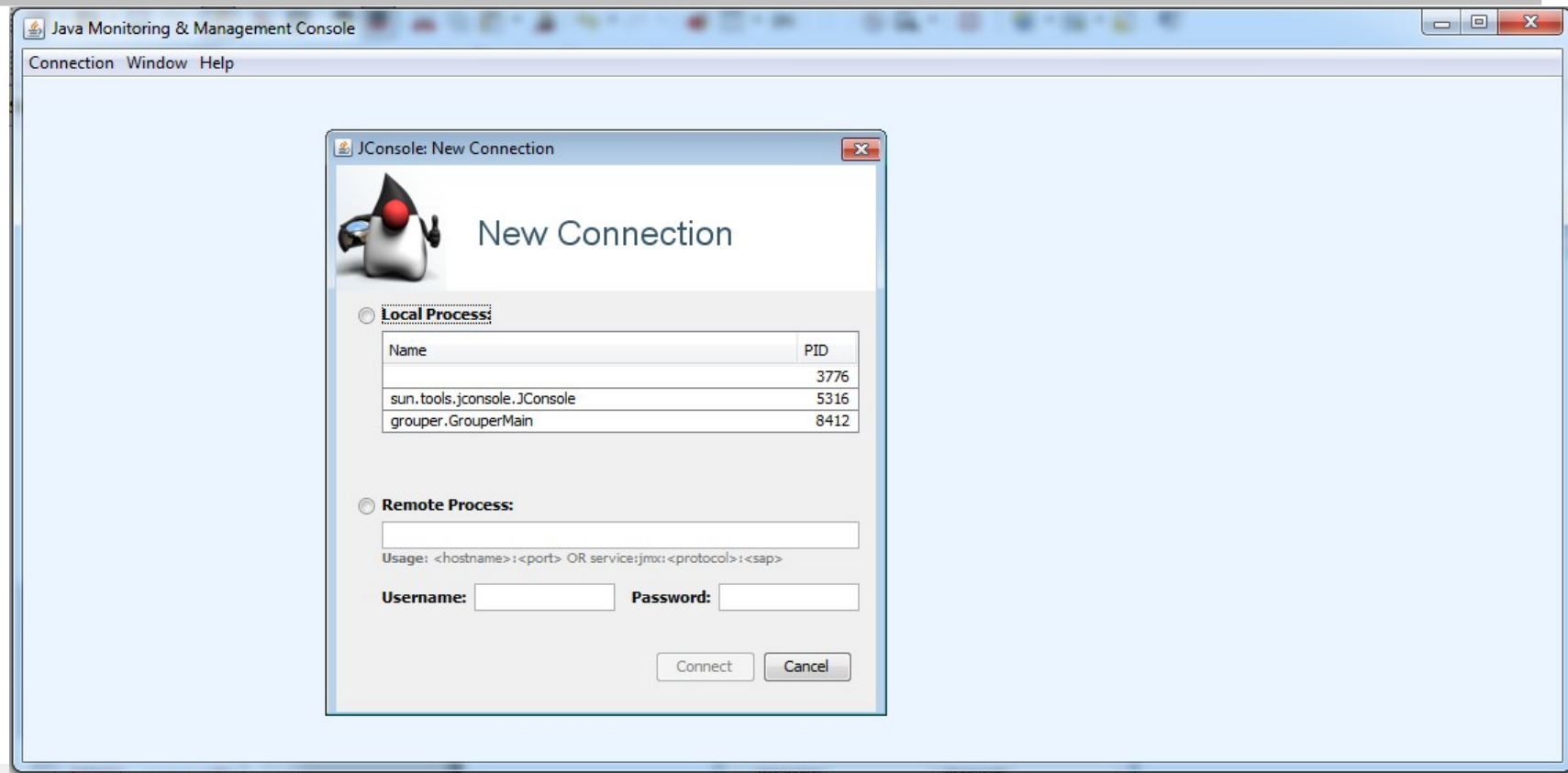
- **jstat & jstatd** – provide information about GC and class loading activities, useful for automated scripting (**jstatd** = RMI daemon):

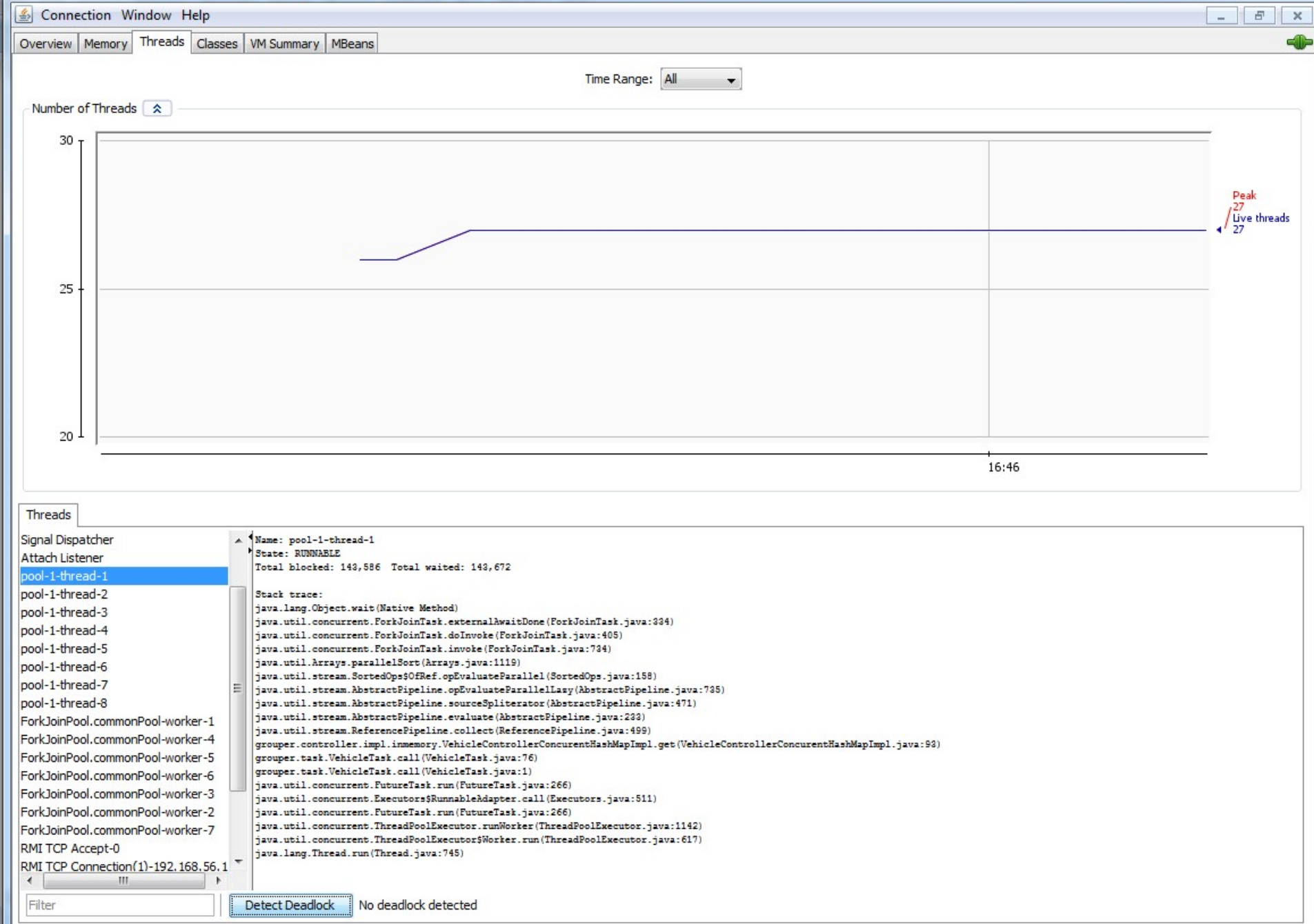
**jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]]]** **Ex: jstat -gc -t -h20 4572 2s**

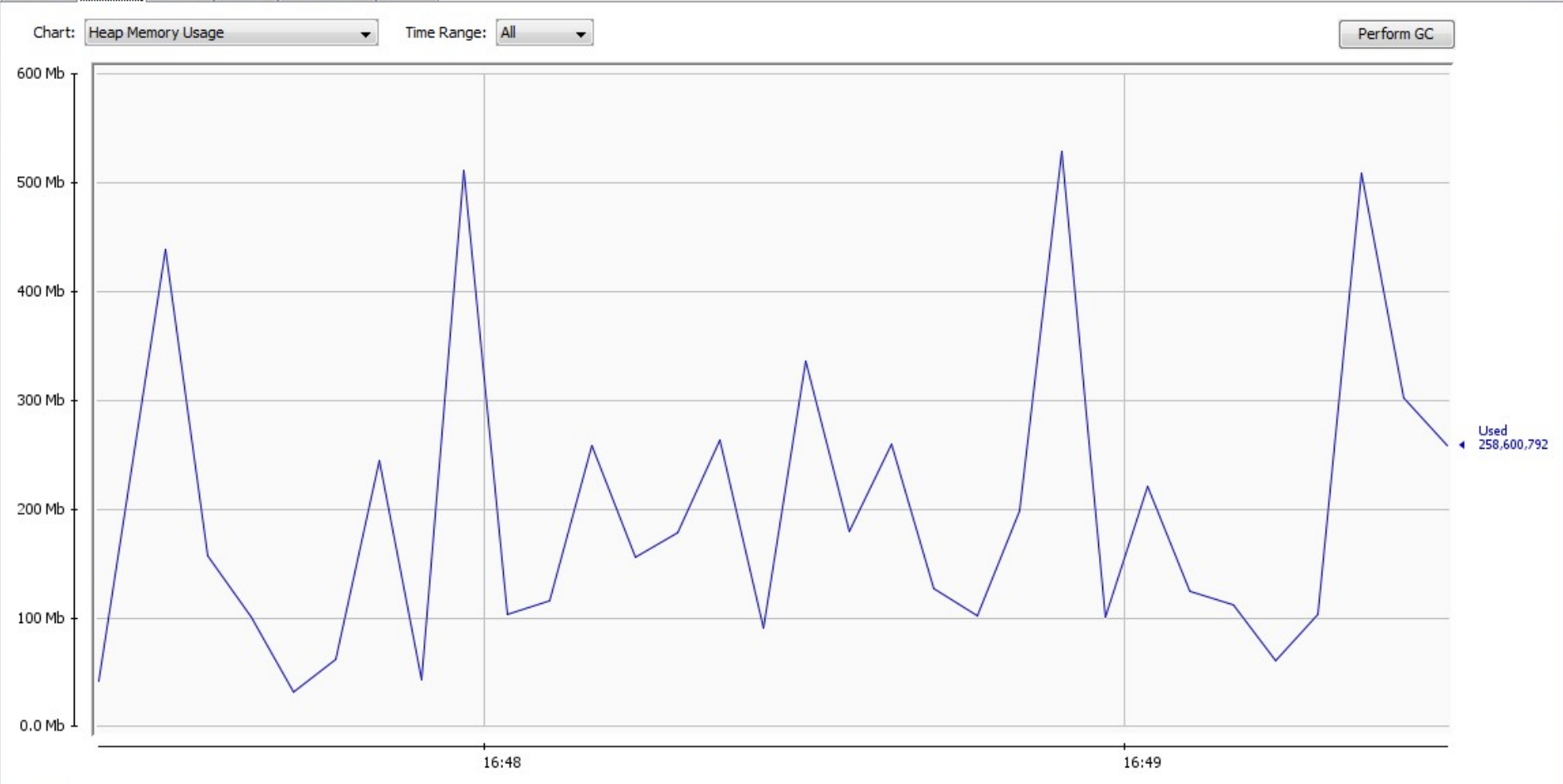
- Statistics options (part of **outputOptions**):
  - class** - statistics on the behavior of the class loader;
  - compiler** - behavior of the HotSpot Just-in-Time compiler;
  - gc** - statistics of the behavior of the garbage collected heap;
  - gccapacity** - capacities of the generations and their spaces;
  - gccause**, **-gcutil** - summary of garbage collection statistics/causes;
  - gcnew**, **-gcnewcapacity**, **-gcold**, **-gcoldcapacity**, **-gcpermcapacity**
    - Young/Old/Permanent generation stats
  - printcompilation** - HotSpot compilation method statistics



## Java GUI tools – JConsole





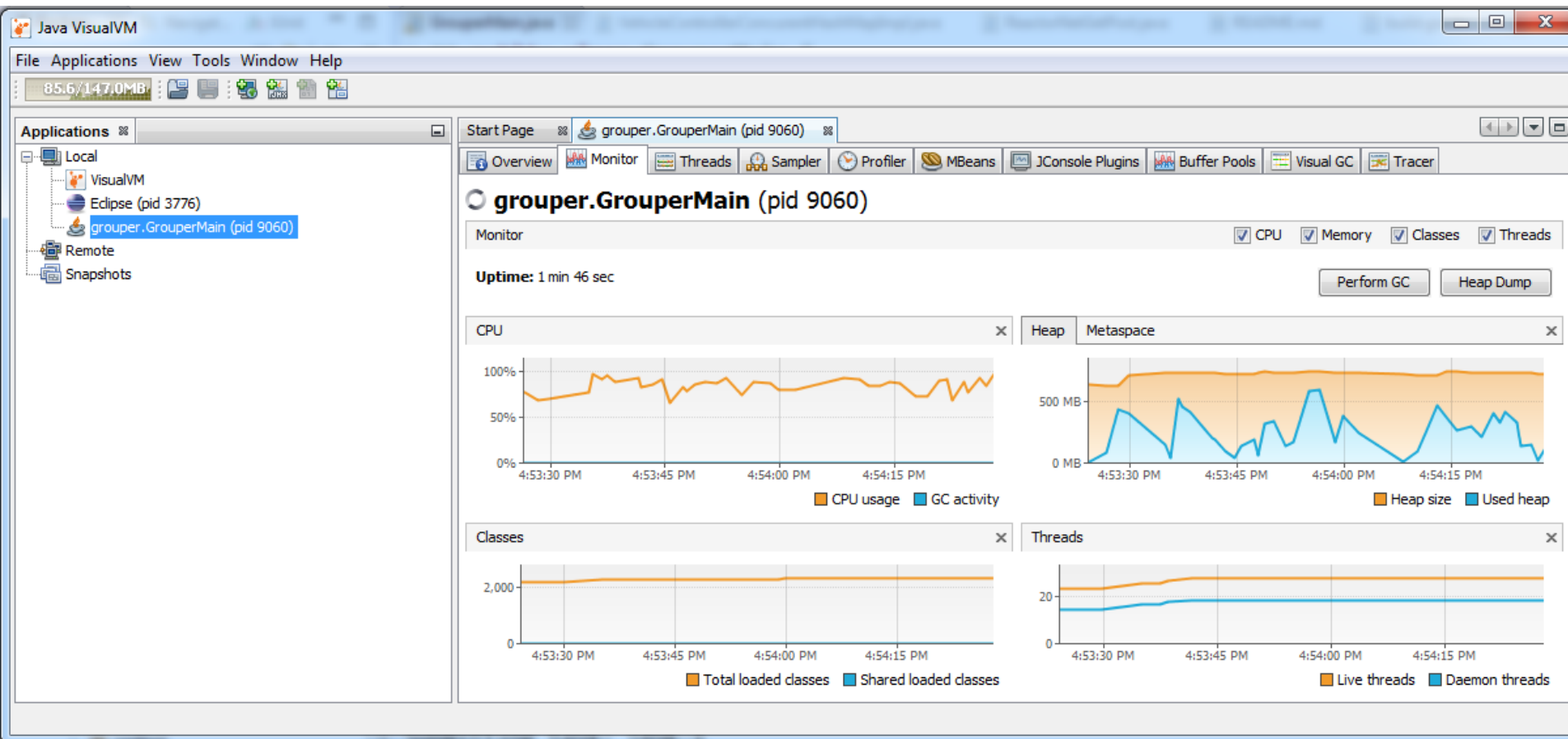


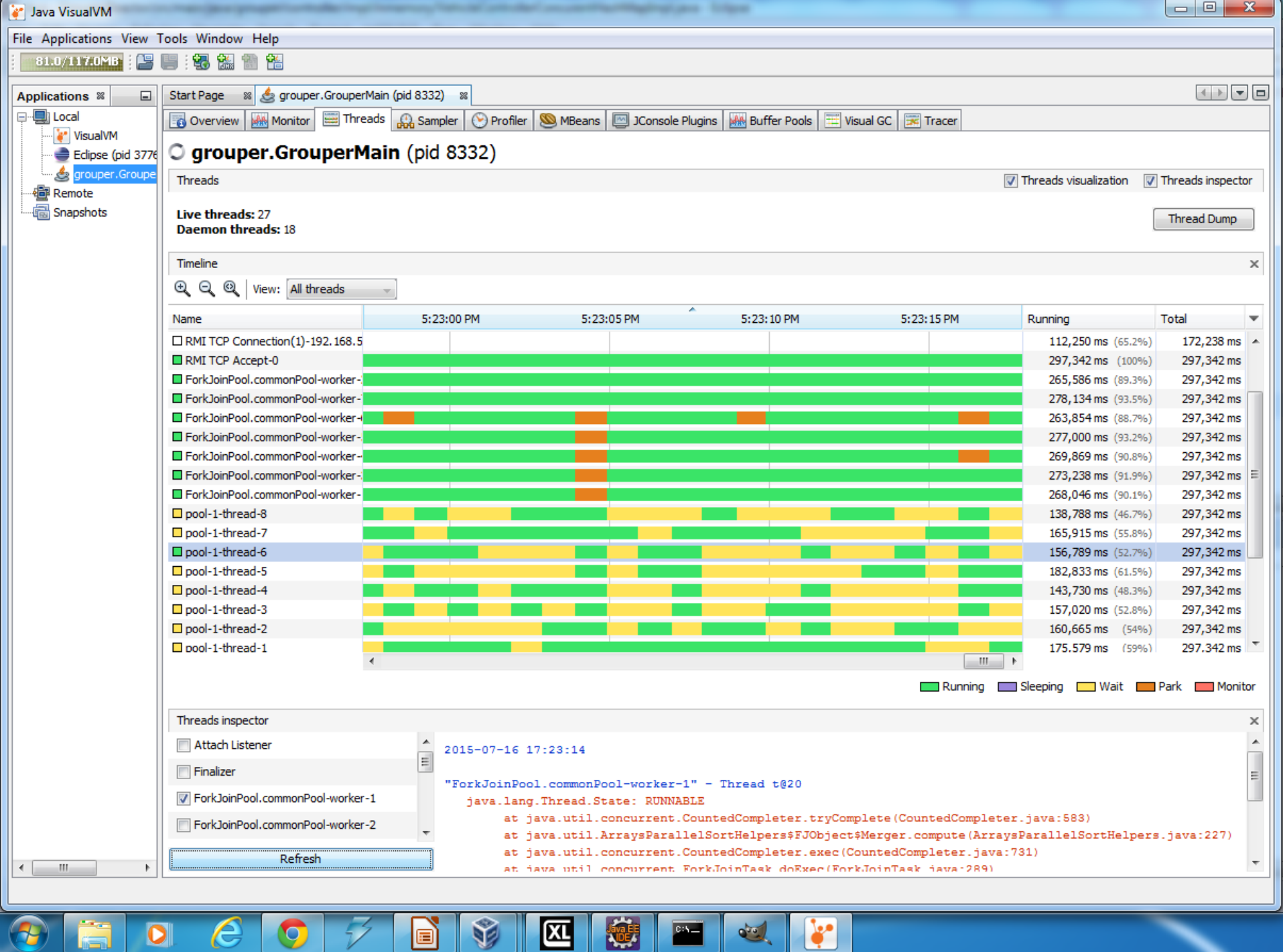
**Details**

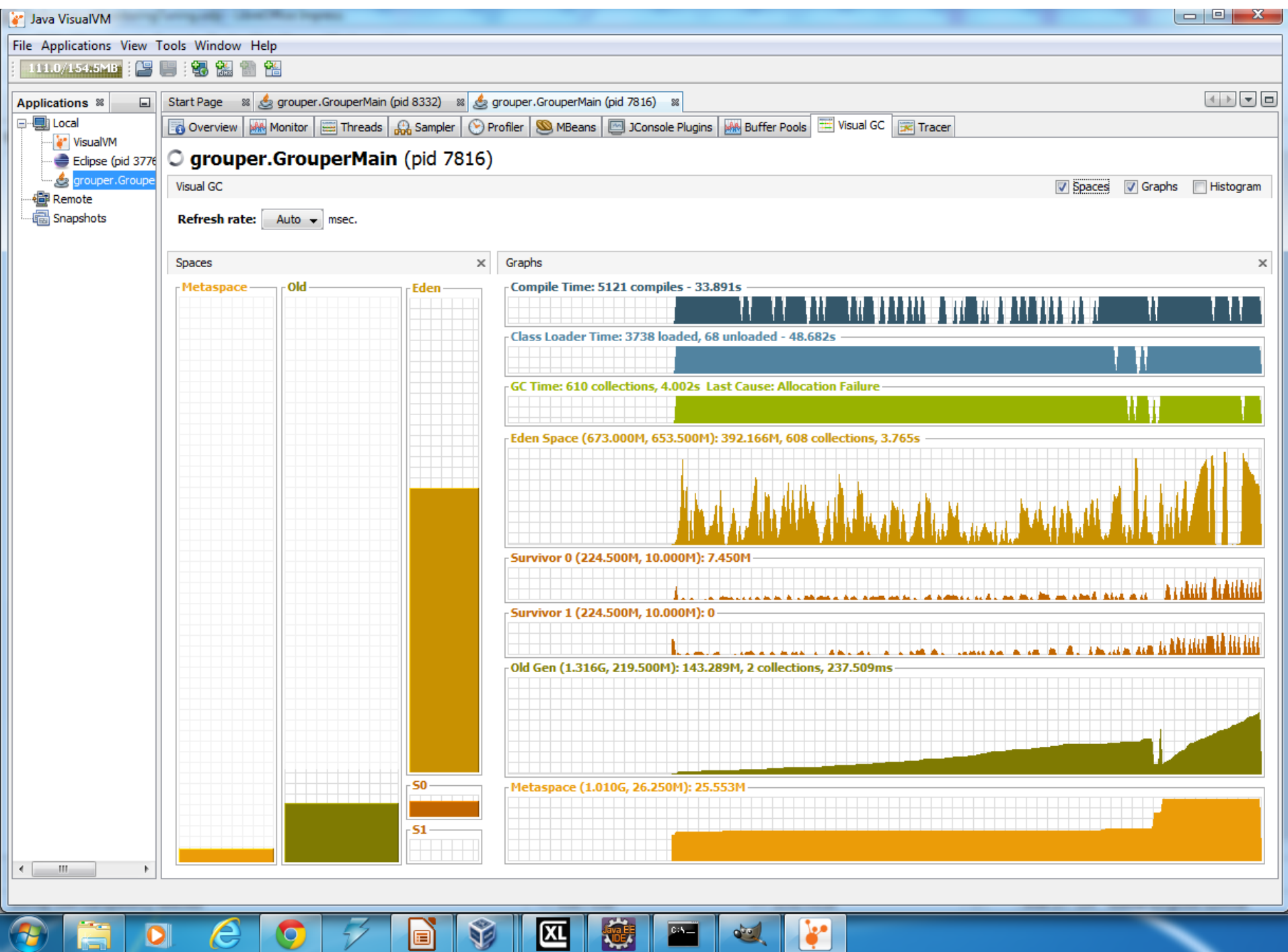
**Time:** 2015-07-16 16:49:30  
**Used:** 295,252 kbytes  
**Committed:** 474,624 kbytes  
**Max:** 1,840,640 kbytes  
**GC time:** 0.000 seconds on PS MarkSweep (0 collections)  
1.143 seconds on PS Scavenge (455 collections)

Category	Usage (%)
Heap	~25%
Non-Heap	~100%

## Java GUI tools – jvisualvm









# Thank's for Your Attention!



**Trayan Iliev**

**CEO of IPT – Intellectual Products  
& Technologies**

<http://iproduct.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>