Effective Software

Lecture 9: JVM - Memory Analysis, Data Structures, Object Allocation

David Šišlák
david.sislak@fel.cvut.cz

JVM Performance Factors and Memory Analysis

» application **performance factors**

- <u>total runtime</u>
  - algorithms (complexity, instructions, synchronization)
  - memory management (garbage collection) overhead
  - data structures (speed of accessing data, cache efficiency, GC pressure)
- <u>memory consumption</u>
  - data structures (memory usage efficiency)

» **memory analysis**

- <u>static memory analysis</u>
  - analyze memory usage at particular time
  - suitable for data structure efficacy analysis, inspect content
- <u>dynamic memory analysis</u>
  - analyze dynamic changes over time
  - suitable for object allocation analysis and memory leak identification

Static Memory Analysis – Object Histogram

» analyze **histogram of objects** – imply global safepoint

- jmap -histo:live {PID}

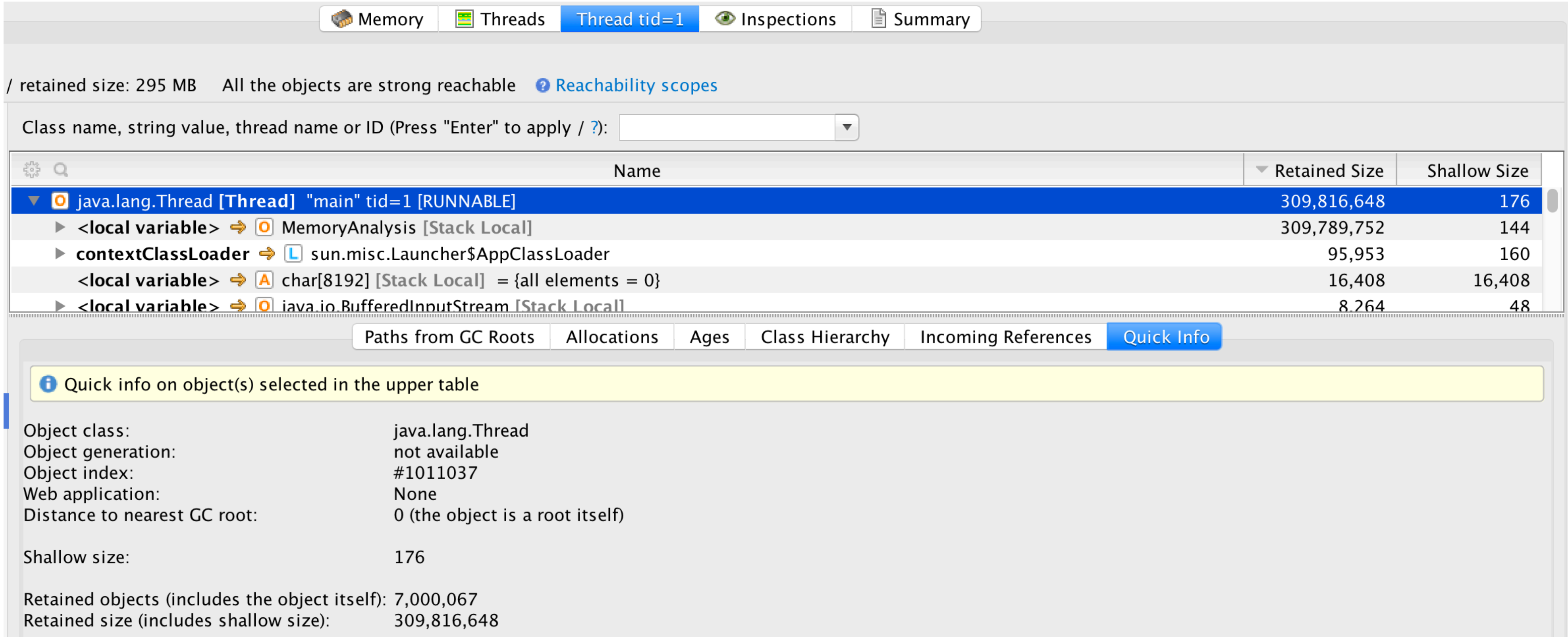
```
 num     #instances         #bytes  class name
----------------------------------------------
   1:       3000257       72006168  java.lang.Integer
   2:       1000122       48005856  java.util.HashMap$Node
   3:       1000012       40000480  java.util.LinkedList$Node
   4:       1000000       40000000  gnu.trove.list.linked.TIntLinkedList$TIntLink
   5:             2       33913088  [D
   6:           162       24963896  [I
   7:       1000001       24000024  java.lang.Double
   8:            26       16781936  [Ljava.util.HashMap$Node;
   9:           604        8056600  [Ljava.lang.Object;
  10:            37        2176208  [B
  11:          1549         184864  [C
  12:           635         109344  java.lang.Class
  13:          1519          48608  java.lang.String
  14:           307          14736  java.util.concurrent.ConcurrentHashMap$Node
  15:           108          12960  java.lang.reflect.Field
  16:            85           8840  java.net.URL
  17:           257           6168  java.lang.Byte
  18:           257           6168  java.lang.Long
  19:           257           6168  java.lang.Short
  20:           123           5904  java.util.Hashtable$Entry
  21:           102           5712  java.lang.ref.SoftReference
  22:             5           4984  [Ljava.util.concurrent.ConcurrentHashMap$Node;
  23:           284           4544  java.lang.Object
```

Static Memory Analysis – Heap Dump

» capture **heap dump** – exported during global safepoint (stop the world)

- -XX:+HeapDumpOnOutOfMemoryError
- jmap -dump:live,format=b,file={name}.hprof {PID}
- jvisualvm, yourkit, …

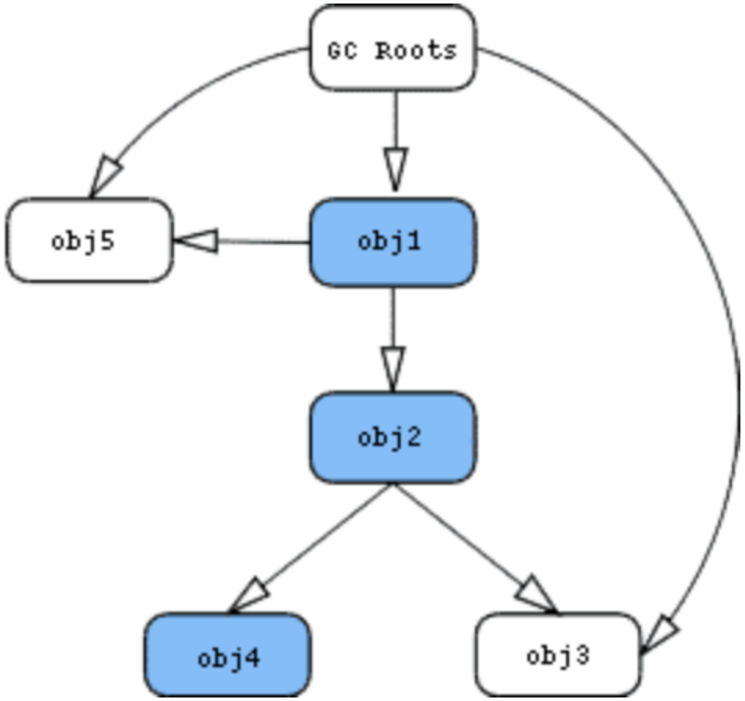
» analyze heap dump – jvisualvm, yourkit, …

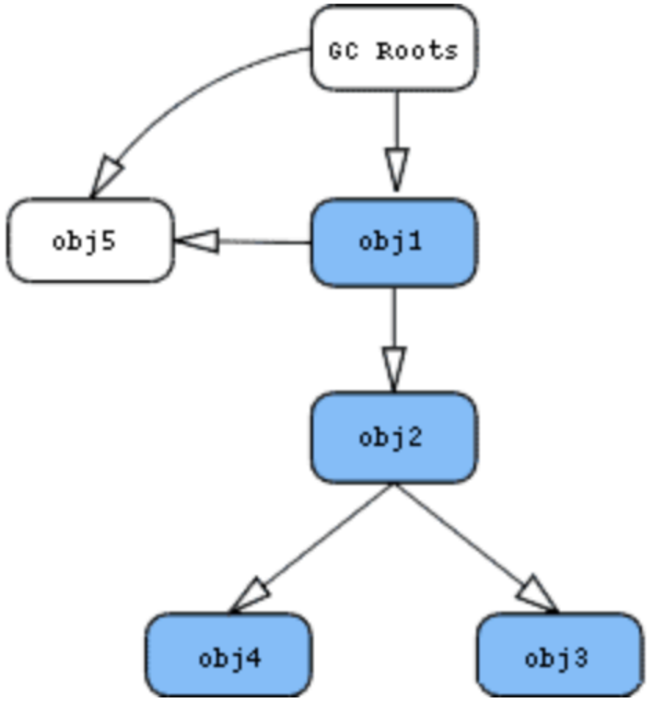


Shallow vs. Retained Size

» **shallow size**
  - memory allocated to store object itself

» **retained size**
  - quantity of memory this object preserves from GC
    - amount of memory freed if the object is GCed
  - own shallow size + shallow size of all objects directly or indirectly accessible **ONLY** from this object





Static Analysis Advanced Inspections

» <u>wasting memory</u>

- **duplicate strings**
  - share string instances via pooling or intern()
- **duplicate objects** – same field contents
  - share them, lazy creation, non-permanent usage
- **zero length arrays**
  - unnecessary load for GC
  - use per-class empty array singleton (e.g. via static field in the class)
- **null fields** - objects having a lot of 'null' fields
  - use subclasses for rarely assigned fields
- **sparse arrays** – big number of 'null', zero or same elements
  - use alternate data structures (e.g. maps or refactor algorithms)
- **inefficient data structure** – large overhead of useless content
  - use different data structures

Static Analysis Advanced Inspections

» memory leak

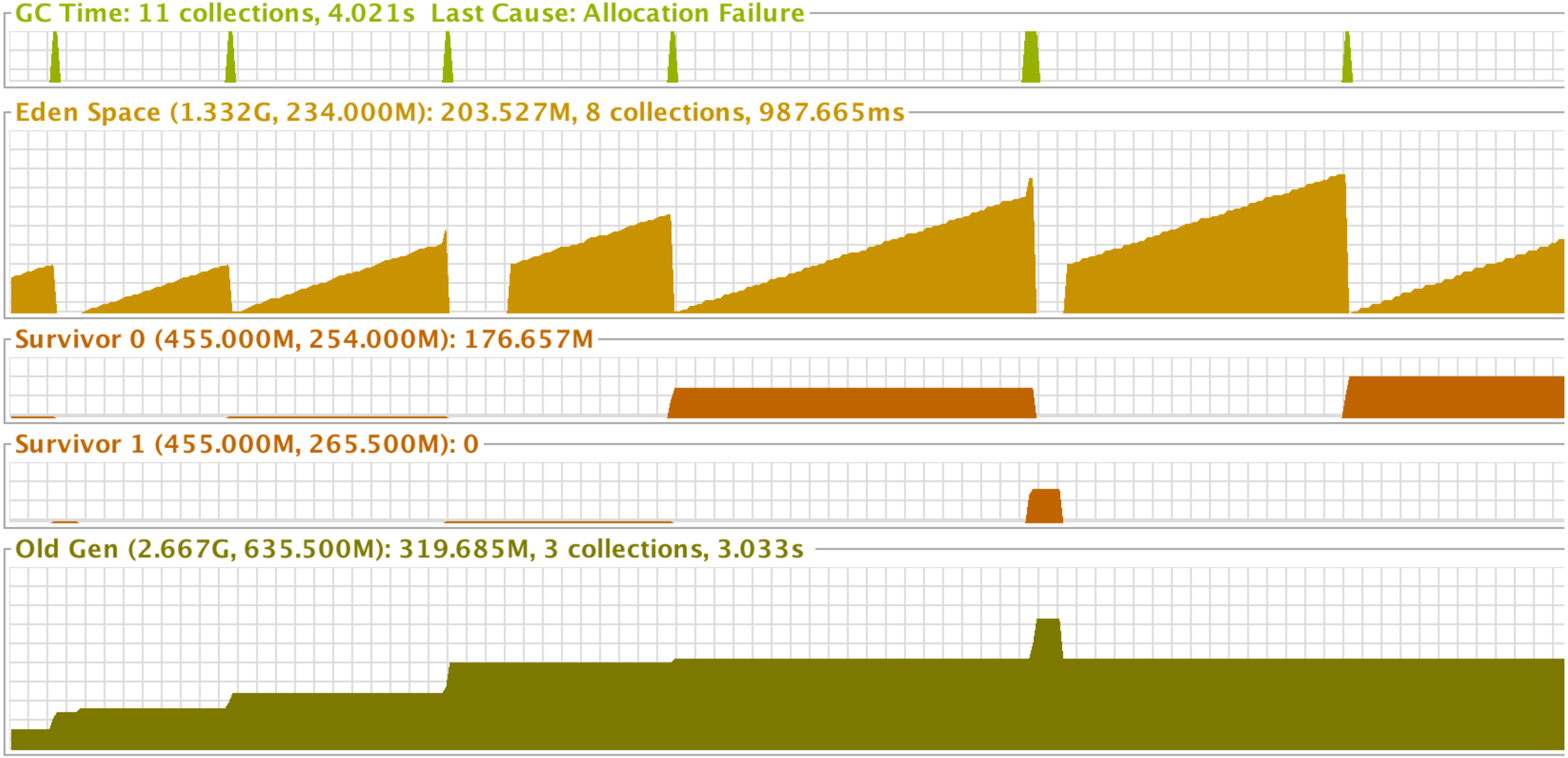
- **object retained from inner class back reference**
  - implicit back reference from inner class instance (even anonymous), e.g. used for callback objects
  - minimize usage of non-static inner class instances

» performance

- **hash tables with non-uniformly distributed hash codes**
  - degraded performance due to hash collisions
  - use better hashCode implementation

Dynamic Memory Analysis – GC Telemetry

» analyze **GC telemetry** – e.g. jvisualvm with VisualGC plugin
  - usage of eden space in time
  - GC collections and their duration
  - not affecting performance of monitored application



Dynamic Memory Analysis – Heap Dumps

» compare **heap dumps**

- difference in object count and size in various application state
- dumps with all objects (not just live) can help analyze object allocations if there is no GC run in between
- each heap dump requires global safepoint (time depends on the heap size)

Old snapshot (the baseline): dump1
New snapshot: dump2 ← **this snapshot**

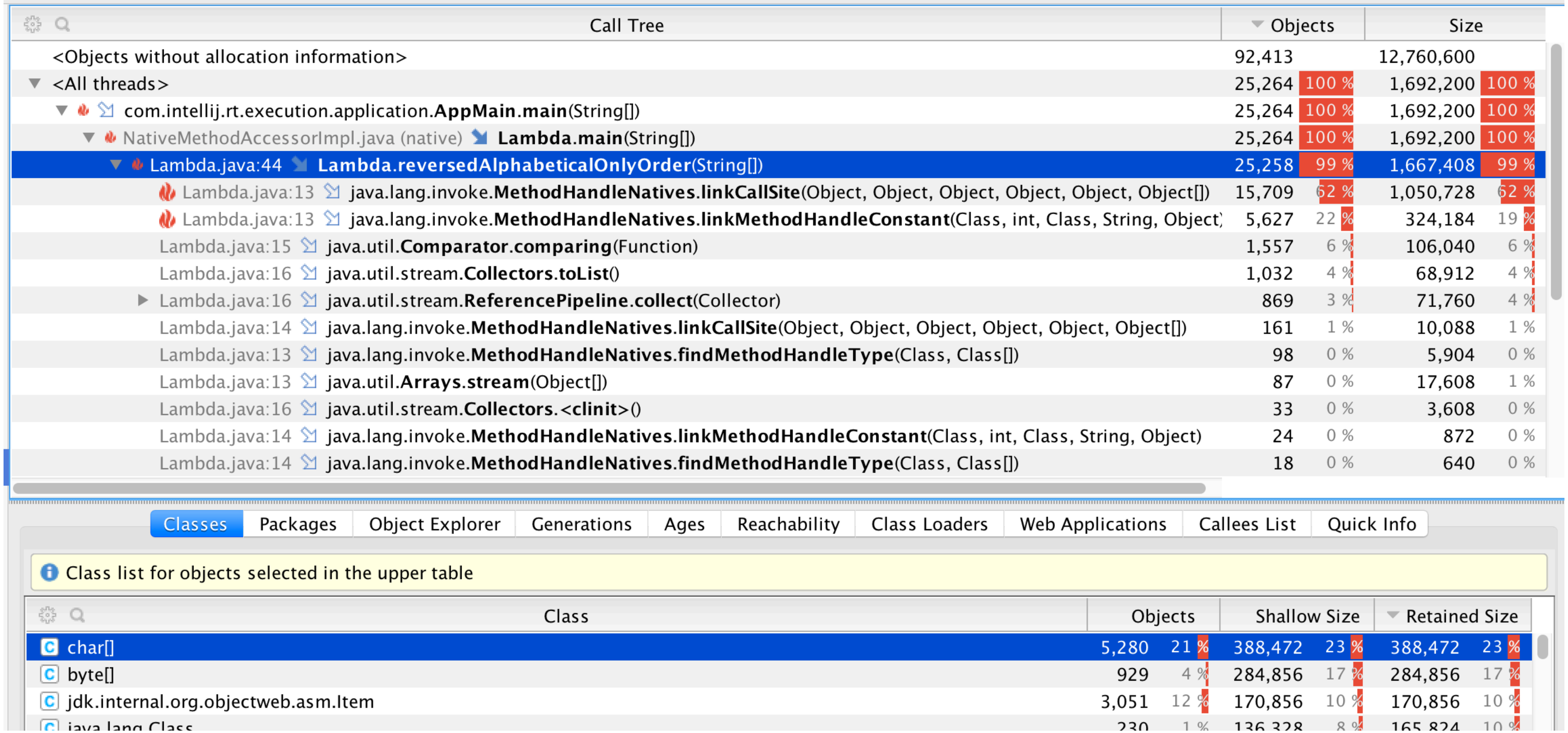
Objects
- Classes
- Classes and packages

Exceptions
- Exceptions

Name	Objects (+/-)		Size (+/-)	
char[]	+5,386	+93 %	+407,040	+6 %
byte[]	+927	+16 %	+292,792	+5 %
jdk.internal.org.objectweb.asm.Item	+3,051	+53 %	+170,856	+3 %
jdk.internal.org.objectweb.asm.Item[]	+174	+3 %	+101,280	+2 %
java.lang.String	+3,219	+55 %	+77,256	+1 %
java.lang.Class[]	+1,275	+22 %	+38,456	+1 %
java.lang.Object[]	+716	+12 %	+34,984	+1 %
jdk.internal.org.objectweb.asm.MethodWriter	+132	+2 %	+29,568	+0 %
java.lang.invoke.MethodType	+729	+13 %	+29,160	+0 %
java.lang.invoke.MethodType$ConcurrentWeakInternSet$WeakEntry	+729	+13 %	+23,328	+0 %
jdk.internal.org.objectweb.asm.Label	+245	+4 %	+15,680	+0 %
java.lang.StringBuilder	+638	+11 %	+15,312	+0 %
jdk.internal.org.objectweb.asm.ClassWriter	+91	+2 %	+15,288	+0 %
java.lang.reflect.Method	+168	+3 %	+14,784	+0 %
jdk.internal.org.objectweb.asm.Type	+439	+8 %	+14,048	+0 %
jdk.internal.org.objectweb.asm.AnnotationWriter	+248	+4 %	+13,888	+0 %
jdk.internal.org.objectweb.asm.ByteVector	+562	+10 %	+13,488	+0 %
java.lang.StringBuffer	+457	+8 %	+10,968	+0 %
jdk.internal.org.objectweb.asm.Frame	+226	+4 %	+10,848	+0 %
java.lang.invoke.MemberName	+326	+6 %	+10,432	+0 %
boolean[]	+34	+1 %	+9,248	+0 %
java.lang.invoke.LambdaForm$Name[]	+130	+2 %	+5,984	+0 %
java.lang.invoke.InvokerBytecodeGenerator	+80	+1 %	+5,120	+0 %
java.lang.invoke.LambdaForm$BasicType[]	+88	+2 %	+4,120	+0 %
java.util.HashMap	+81	+1 %	+3,888	+0 %
jdk.internal.org.objectweb.asm.Type[]	+114	+2 %	+3,880	+0 %
java.util.AbstractList$Itr	+115	+2 %	+3,680	+0 %
java.util.HashMap$ValueIterator	+81	+1 %	+3,240	+0 %

Dynamic Memory Analysis – Allocation Tracking

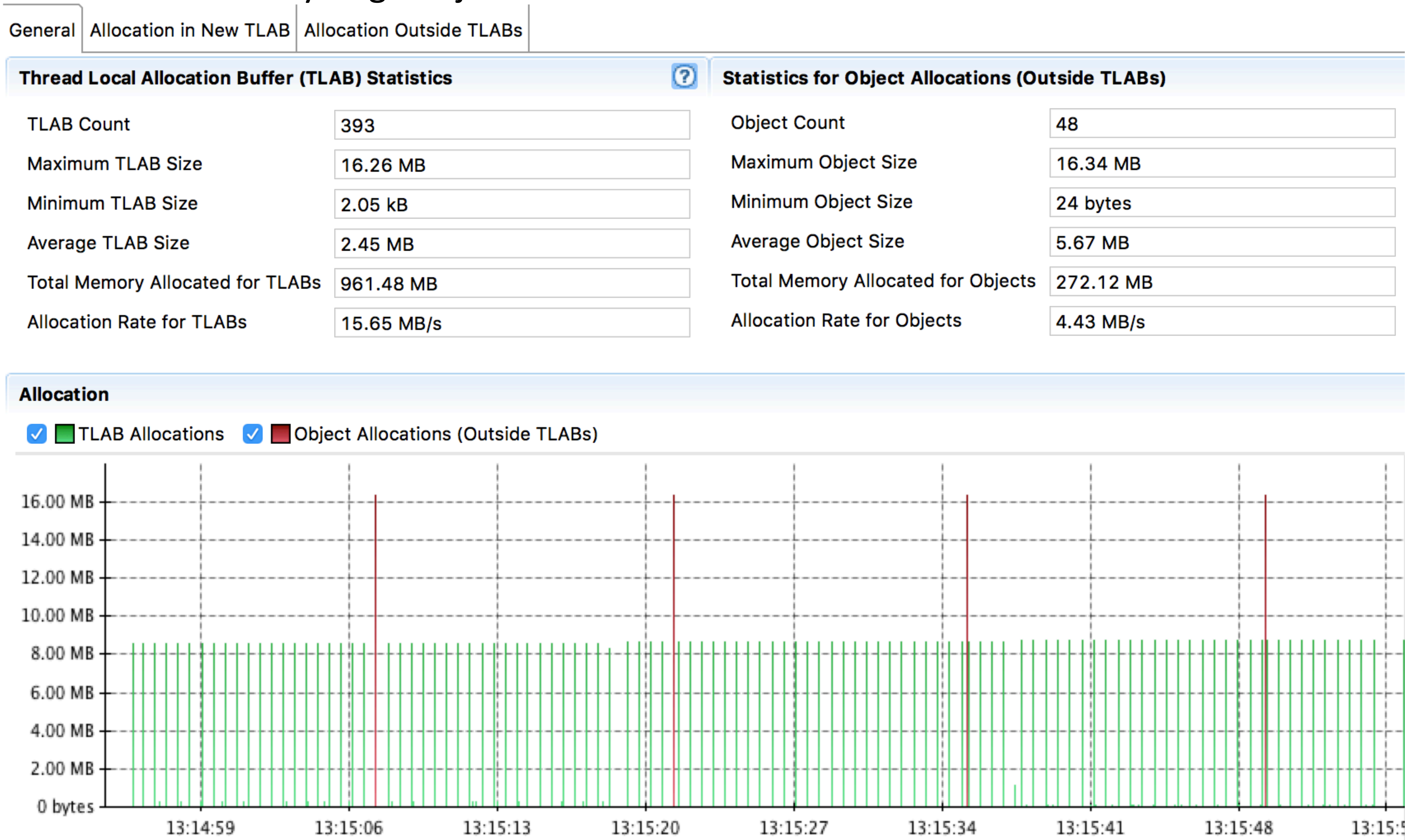
- allocation tracking - **memory profiler**
  - track every n-th object allocation (trade-off between precision and speed)
  - affect performance of profiled application, injects **traceObjAlloc** byte code
    - introduce a lot of byte code + consume memory
    - decreases possibility of JIT optimizations



Dynamic Memory Analysis – Allocation Tracking

» allocation tracking – **flight recording** using jmc – <u>no byte code instrumentation</u>

- identify large object allocations outside TLAB



Dynamic Memory Analysis – Allocation Tracking

» allocation tracking – **flight recording** using jmc – <u>no byte code instrumentation</u>

- identify large object allocations outside TLAB

General | Allocation in New TLAB | Allocation Outside TLABs

Allocation by Class | Allocation by Thread | Allocation Profile

Allocation Pressure

Class	Average Object Siz	TLABs	Total TLAB Size
java.lang.Integer	24 bytes	38	325.76 MB
java.util.HashMap$Node	48 bytes	39	306.85 MB
java.lang.Double	24 bytes	15	128.52 MB
gnu.trove.list.linked.TIntLinkedList$TIntLink	40 bytes	14	119.89 MB
java.util.LinkedList$Node	40 bytes	5	43.15 MB
java.lang.Object[]	159.71 kB	49	21.43 MB
java.io.ObjectStreamClass$WeakClassKey	56 bytes	35	2.12 MB
char[]	128 bytes	40	1.94 MB
java.lang.Class	160 bytes	1	1.82 MB

Stack Trace

Stack Trace	TLABs	Total TLAB Size	Pressure
java.util.ArrayList.<init>(int)	1	16.26 MB	75.89%
java.util.AbstractCollection.toArray()	1	1.82 MB	8.49%
java.io.ObjectOutputStream.defaultWriteFields(Object, ObjectStreamClass)	17	1.32 MB	6.17%
java.io.ObjectOutputStream$HandleTable.growEntries()	11	1.27 MB	5.94%
sun.management.MemoryUsageCompositeData.getCompositeData()	4	171.98 kB	0.78%
sun.reflect.misc.MethodUtil.invoke(Method, Object, Object[])	3	164.10 kB	0.75%
java.io.ObjectStreamClass.invokeWriteObject(Object, ObjectOutputStream)	2	71.89 kB	0.33%
java.io.ObjectInputStream$HandleTable.<init>(int)	1	64.12 kB	0.29%
java.lang.reflect.Array.newInstance(Class, int)	1	64.05 kB	0.29%
sun.management.MappedMXBeanType$MapMXBeanType.toOpenTypeData(Object)	1	52.48 kB	0.24%

Dynamic Memory Analysis – Allocation Tracking

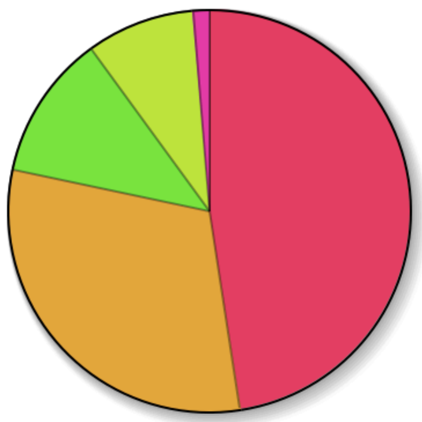
» allocation tracking – **flight recording** using jmc – no byte code instrumentation

- identify large object allocations outside TLAB

General | Allocation in New TLAB | Allocation Outside TLABs

Allocation by Class | Allocation by Thread | Allocation Profile

Allocation Pressure



Class	Objects	Total Size	Pressure
double[]	8	129.37 MB	47.54%
int[]	14	83.76 MB	30.78%
java.util.HashMap$Node[]	2	32.00 MB	11.76%
java.lang.Object[]	10	22.90 MB	8.42%
byte[]	4	4.09 MB	1.50%
java.util.TreeMap$Entry	4	256 bytes	0.00%
java.util.TreeMap	2	160 bytes	0.00%
char[]	1	128 bytes	0.00%
java.util.TreeMap$KeyIterator	1	56 bytes	0.00%
java.lang.Integer	1	24 bytes	0.00%

Stack Trace

Stack Trace	Objects	Total Size	Pressure
gnu.trove.impl.hash.TIntDoubleHash.setUp(int)	4	32.68 MB	39.02%
it.unimi.dsi.fastutil.ints.Int2DoubleOpenHashMap.<init>(int, float)	4	32.00 MB	38.20%
it.unimi.dsi.fastutil.ints.IntArrayList.<init>(int)	4	15.26 MB	18.22%
gnu.trove.list.array.TIntArrayList.<init>(int, int)	1	3.81 MB	4.55%
java.io.ObjectOutputStream$HandleTable.growEntries()	1	1.40 kB	0.00%

Data Structures – Primitives and Objects

» **primitives**: boolean(1), byte(1), char(2), int(4), long(8), float(4), double(8)
- without implicit allocation
- stored in variables or operand stack in frame

» **objects** (object header structure overhead) allocated on the heap
- every object is descendant of Object by default
  - methods – clone(), equals, getClass(), hashCode(), wait(…), notify(…), finalize()
- objects for primitives: Boolean, Byte, Character, Integer, Long, Float, Double; can be **null**
- objects with multiple fields use **type group alignment** and padding:
  - longs and doubles (8B)
  - ints and floats (4B)
  - shorts and chars (2B)
  - bytes and booleans
  - references (4B / 8B)

Object structure (64-bit JVM):
- header 12 or 16 Bytes
- object data super class first

8B - mark word
4B / 8B – Klass ref.
… object data

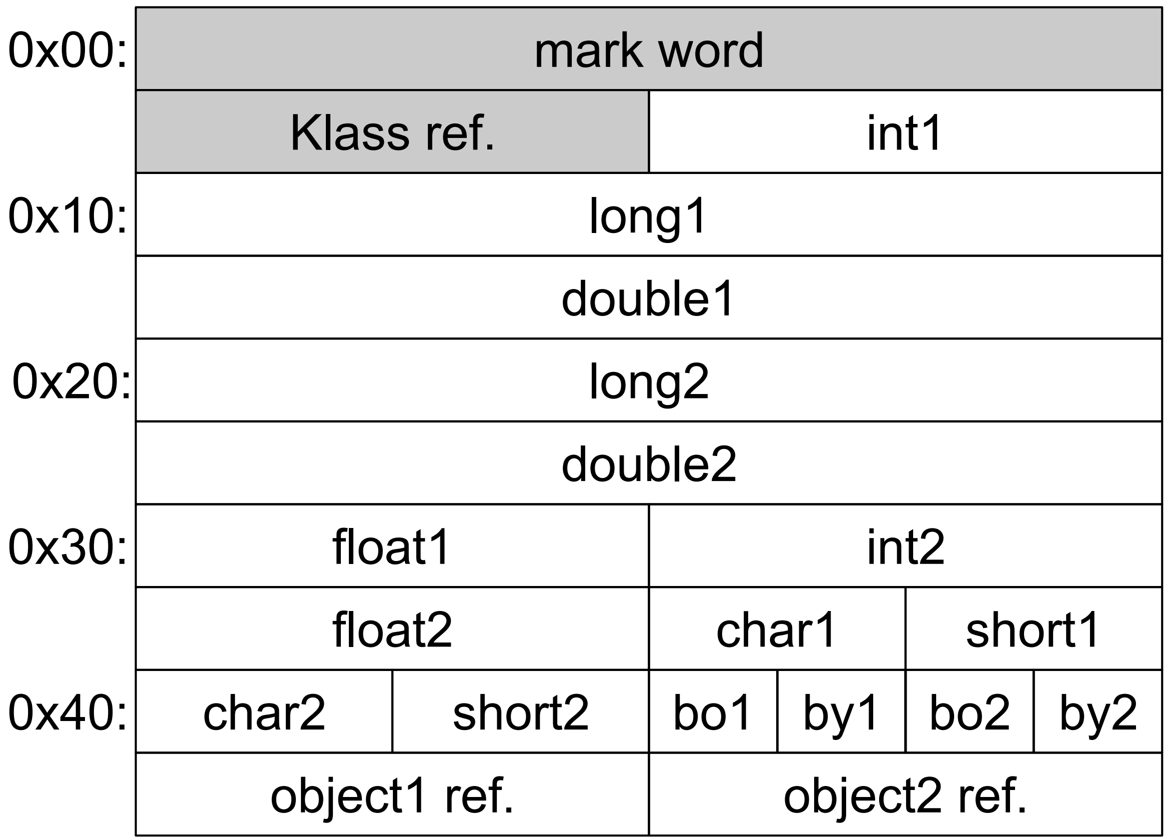
Data Structures – Object Example 64-bit <32GB Heap

```java
class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte byte2;
    private char char2;
    private short short2;
    private int int2;
    private long long2;
    private float float2;
    private double double2;
    private Object object2;

    Structure(int value, Object ref

    @Override
    public String toString() {...}
}
```

Object structure (64-bit JVM) **using compressed OOP**:
- object size 80 Bytes



Data Structures – Object Example 64-bit >=32GB Heap

```java
class Structure {
    private boolean boolean1;
    private byte byte1;
    private char char1;
    private short short1;
    private int int1;
    private long long1;
    private float float1;
    private double double1;
    private Object object1;
    private boolean boolean2;
    private byte byte2;
    private char char2;
    private short short2;
    private int int2;
    private long long2;
    private float float2;
    private double double2;
    private Object object2;

    Structure(int value, Object ref

    @Override
    public String toString() {...}
}
```

<u>Object structure (64-bit JVM) **using standard OOP**:</u>

- object size 96 Bytes

Address								
0x00:	mark word							
	Klass ref.							
0x10:	long1							
	double1							
0x20:	long2							
	double2							
0x30:	int1				float1			
	int2				float2			
0x40:	char1		short1		char2		short2	
	bo1	by1	bo2	by2	empty padding			
0x50:	object1 ref.							
	object2 ref.							

Data Structures – Arrays

» **single-dimension arrays**

- special data structure which store a number of items of the same type in linear order; have the defined limit
- JAVA automatically check limitations
- allocated on the heap
- primitives – keep primitive values directly
- objects – keep references to objects (4B or 8B references)

» **multi-dimensional arrays**

- arrays of arrays - ragged array (non-uniform sub-level lengths)
- slower access due to dereferencing (multiple memory read operations) and multi index bound checks
- consider **flatten array**

<u>Array object structure (64-bit JVM):</u>

- header 16 or 20 Bytes
- sequence of array values

8B - mark word
4B / 8B – Klass ref.
4B – array length
sequence of values

Memory Efficiency – Primitive Objects

» **memory efficiency**

$$\frac{useful\_content\_size}{retained\_size} * 100[\%]$$

» correlates with **cache efficacy**

- all others data optionally in different cache lines (64 B) are read as well

» **data locality** further speed-up processing utilizing already cached data

Object	Useful size	Retained size (Efficiency) <32GB heap	Retained size (Efficiency) >=32GB heap
Boolean	1 bit	16 B (0.78 %)	24 B (0.52 %)
Byte	1 B	16 B (6.25 %)	24 B (4.17 %)
Short, Character	2 B	16 B (12.50 %)	24 B (8.34 %)
Integer, Float	4 B	16 B (25.00 %)	24 B (16.67 %)
Long, Double	8 B	24 B (33.34 %)	24 B (33.34 %)

Primitive Objects

» **auto boxing** and **un-boxing** during assignment and parameter passing

- valueOf({primitive}) and {primitve}Value() methods

» all primitive objects are **immutable** (final values)

» beware of **inefficiencies** caused by boxing and un-boxing

```
public class PrimitiveObject {
    private static Integer integer = 0;

    public static void main(String[] args) {
        integer++;
    }
}
```

```
static {};
  Code:
     0: iconst_0
     1: invokestatic  #4
     4: putstatic     #2
     7: return
```

```
private static Integer integer = 0;

// Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
// Field integer:Ljava/lang/Integer;
```

```
public static void main(java.lang.String[]);
  Code:
     0: getstatic     #2
     3: invokevirtual #3
     6: iconst_1
     7: iadd
     8: invokestatic  #4
    11: putstatic     #2
    14: return
```

```
integer++;
// Field integer:Ljava/lang/Integer;
// Method java/lang/Integer.intValue:()I


// Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
// Field integer:Ljava/lang/Integer;
```

Conversion Inefficiencies - Example

» count word histogram

```
public static void main(String[] args) {
    Map<String, Integer> m = new TreeMap<String, Integer>();
    for (String word : args) {
        Integer freq = m.get(word);
        m.put(word, (freq == null ? 1 : freq + 1));
    }
    System.out.println(m);
 }
```

Conversion Issues - Example

```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

Conversion Issues - Example

```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

true	true
true	false
true	true

Note: after serialization the second is always false

Primitive Object – Identity Semantics

- **identity semantics** using cache for **valueOf({primitive})**
  - Short, Integer, Long – caches <-128;+127>
  - Byte – caches all values
  - Character – caches <0;+127>
- **not working** for objects created by constructor (e.g. new Integer(1))

```java
private static class ShortCache {
    private ShortCache(){}

    static final Short cache[] = new Short[-(-128) + 127 + 1];

    static {
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Short((short)(i - 128));
    }
}
public static Short valueOf(short s) {
    final int offset = 128;
    int sAsInt = s;
    if (sAsInt >= -128 && sAsInt <= 127) { // must cache
        return ShortCache.cache[sAsInt + offset];
    }
    return new Short(s);
}

public short shortValue() {
    return value;
}
```

Memory Efficiency – Java Collections

» **LinkedList<E>**
  - uses Node<E> object with bi-directional links

» **ArrayList<E>**
  - backend elementData array with references to objects

» **HashMap<K,V>**
  - backend hash table of Node<K,V> with cached hashCode and linked collisions

Note: Measured for 1 million of elements in Collections and Map

Object	Useful size	Retained size (Efficiency) <32GB heap	Retained size (Efficiency) >=32GB heap
LinkedList<Integer>	4 MiB	34.33 MiB (11.65 %)	47.26 MiB (8.46 %)
ArrayList<Integer>	4 MiB	17.73 MiB (22.56 %)	25.72 MiB (15.55 %)
HashMap<Integer,Double>	12 MiB	70.19 MiB (17.10 %)	87.67 MiB (13.69 %)

Collections for Performance

» **Trove** – Lesser GNU Public License (LGPL)

» **FastUtil** – Apache License 2.0

» **collections for performance**

- type-specific maps, sets, lists and queues
  - remove overheads related auto-boxing and un-boxing
- small memory footprint
  - much better caching
  - sequential access is very fast
- fast access and insertion
- use **open addressing** hashing in Maps instead of chaining approch
- support big collections (>2^{31} elements) in *FastUtil*
- support custom hashing strategies in *Trove*

```
char[] foo, bar;
foo = new char[] {'a','b','c'};
bar = new char[] {'a','b','c'};
System.out.println(foo.hashCode() == bar.hashCode() ? "equal" : "not equal");
System.out.println(foo.equals(bar) ? "equal" : "not equal");
```

Open Addressing Hash Table

» **eliminates** the need for Map.Entry<K,V> **wrapper** supporting chaining
  - typed keys & values arrays
  - state byte array – FREE, FULL, REMOVED (*Trove*, total 3 arrays)
  - special 0/null key tracking + default return value for empty (*FastUtil*, total 2 arrays)

» smaller load factor implies less conflicts (*Trove* 0.5, *FastUtil* 0.75)

» **collision resolution** scheme
  - linear probing (*FastUtil*) – better cache utilization due to data locality
  - double hash probing (Trove) – less conflicts

$$h(i,k) = (h_1(k) + i \cdot h_2(k)) \bmod |T|$$

    - h_2 cannot be 0, thus +1 is usually used

» complex **deletion** to keep conflict searching consistent
  - shift last collision element instead of removed (*FastUtil*)
  - keep removed elements – used by later puts (*Trove*)

» usage of **prime number** size of hash table reduce hashing collisions (*Trove*)

» usage of **power of two** size of hash table leads to fast bit operations (*FastUtil*)

Memory Efficiency – Collections for Performance

<u>Note:</u> 1 million of elements stored

Object	Useful size	Retained size (Efficiency) <32GB heap	Retained size (Efficiency) >=32GB heap
LinkedList<Integer>	4 MiB	34.33 MiB (11.65 %)	47.26 MiB (8.46 %)
TIntLinkedList *(Trove)*	4 MiB	20.60 MiB (19.42 %)	24.54 MiB (13.54 %)
ArrayList<Integer>	4 MiB	17.73 MiB (22.56 %)	25.72 MiB (15.55 %)
TIntArrayList *(Trove)*	4 MiB	~4.00 MiB (~100.00 %)	~4.00 MiB (~100.00%)
IntArrayList *(FastUtil)*	4 MiB	~4.00 MiB (~100.00 %)	~4.00 MiB (~100.00%)
HashMap<Integer,Double>	12 MiB	70.19 MiB (17.10 %)	87.67 MiB (13.69 %)
TIntDoubleHashMap *(Trove)*	12 MiB	27.85 MiB (43.09 %)	27.85 MiB (43.09 %)
Int2DoubleOpenHashMap *(FastUtil)*	12 MiB	25.17 MiB (47.68 %)	25.17 MiB (47.68 %)

Collection Resizing – Default Expected Capacity

- **run-time inefficiencies** caused by collection resizing
  - explicitly specify expected collection capacity
- **ArrayList**
  - shared static default empty backend array
  - backend array default capacity 10 (allocated during first add)
  - grow implies copy of all previous elements - strategy +~50%
  - no automatic shrinking, manual using trimToSize
- **TIntArrayList** (Trove)
  - backend array default capacity 10 (allocated immediately)
  - grow implies copy of all previous elements - strategy *2
  - no automatic shrinking, manual using trimToSize
- **IntArrayList** (FastUtil)
  - backend array default capacity 16 (allocated immediately)
  - grow implies copy of all previous elements - strategy *2
  - no automatic shrinking, manual using trim

Collection Resizing – Default Expected Capacity

- **HashMap**
  - hash table initialized with the first element
  - default hash table size 16 (default load factor 0.75)
    - custom capacity rounded to power of two
  - grow implies re-hashing (iteration + puts) of all previous elements
    - strategy *2
  - hash table shrinking not supported at all
- **TIntDoubleHashMap** (Trove)
  - default hash table size 23 (default load factor 0.5)
    - custom capacity adjusted to nearest bigger prime number
  - grow implies re-hashing (iteration + puts) of all previous elements
    - strategy nearest bigger prime number for size * 2
  - auto compaction after certain number of removals
    - nearest bigger prime number for the currently stored elements
    - can be temporarily disabled if you are planning to do a lot of removals

Collection Resizing – Default Expected Capacity

» **Int2DoubleOpenHashMap**

- backend arrays allocated immediately
- default hash table size 16 (default load factor 0.75)
  - custom capacity rounded to power of two
- grow implies re-hashing (iteration + puts) of all previous elements
  - strategy *2
- auto shrinking after remove if used less than ¼ - strategy :2
  - not shrinking under minimum hash table size 16

» **further optimizations** possible

- use stubs for no/one element collections when your application contains a lot of collections