Effective Software

Lecture 10: JVM - Memory Analysis, Data Structures, Collections for Performance

David Šišlák

[Email](mailto:david.sislak@fel.cvut.cz)
JVM Performance Factors and Memory Analysis

» application **performance factors**
  • **total runtime**
    – algorithms (complexity, instructions, synchronization)
    – memory management (garbage collection) overhead
    – **data structures** (speed of data access, cache efficiency, GC pressure)
  • **memory consumption**
    – **data structures** (memory usage efficiency)

» **memory analysis**
  • **static memory analysis**
    – analyze memory usage at particular time
    – suitable for data structure efficacy analysis, inspect content
  • **dynamic memory analysis**
    – 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 (stop the world)
  - `jmap -histo:live {PID}`

<table>
<thead>
<tr>
<th>num</th>
<th>instances</th>
<th>#bytes</th>
<th>class name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30000257</td>
<td>72006168</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>2</td>
<td>1000122</td>
<td>48005856</td>
<td>java.util.HashMap$Node</td>
</tr>
<tr>
<td>3</td>
<td>1000012</td>
<td>40000480</td>
<td>java.util.LinkedList$Node</td>
</tr>
<tr>
<td>4</td>
<td>10000000</td>
<td>400000000</td>
<td>gnu.trove.list.linked.TIntLinkedList$TIntLink</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>33913088</td>
<td>[D</td>
</tr>
<tr>
<td>6</td>
<td>162</td>
<td>24963896</td>
<td>[I</td>
</tr>
<tr>
<td>7</td>
<td>1000001</td>
<td>24000024</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td>8</td>
<td>26</td>
<td>16781936</td>
<td>[Ljava.util.HashMap$Node;</td>
</tr>
<tr>
<td>9</td>
<td>604</td>
<td>8056600</td>
<td>[Ljava.lang.Object;</td>
</tr>
<tr>
<td>10</td>
<td>37</td>
<td>2176208</td>
<td>[B</td>
</tr>
<tr>
<td>11</td>
<td>1549</td>
<td>184864</td>
<td>[C</td>
</tr>
<tr>
<td>12</td>
<td>635</td>
<td>109344</td>
<td>java.lang.Class</td>
</tr>
<tr>
<td>13</td>
<td>1519</td>
<td>48608</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>14</td>
<td>307</td>
<td>14736</td>
<td>java.util.concurrent.ConcurrentHashMap$Node</td>
</tr>
<tr>
<td>15</td>
<td>108</td>
<td>12960</td>
<td>java.lang.reflect.Field</td>
</tr>
<tr>
<td>16</td>
<td>85</td>
<td>8840</td>
<td>java.net.URL</td>
</tr>
<tr>
<td>17</td>
<td>257</td>
<td>6168</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>18</td>
<td>257</td>
<td>6168</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td>19</td>
<td>257</td>
<td>6168</td>
<td>java.lang.Short</td>
</tr>
<tr>
<td>20</td>
<td>123</td>
<td>5904</td>
<td>java.util.HashMap$Entry</td>
</tr>
<tr>
<td>21</td>
<td>102</td>
<td>5712</td>
<td>java.lang.ref.SoftReference</td>
</tr>
<tr>
<td>22</td>
<td>5</td>
<td>4984</td>
<td>[Ljava.util.concurrent.ConcurrentHashMap$Node;</td>
</tr>
<tr>
<td>23</td>
<td>284</td>
<td>4544</td>
<td>java.lang.Object</td>
</tr>
</tbody>
</table>
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, ...`

```
retained size: 295 MB   All the objects are strong reachable

Class name, string value, thread name or ID (Press "Enter" to apply / ?):   

<table>
<thead>
<tr>
<th>Name</th>
<th>Retained Size</th>
<th>Shallow Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;local variable&gt; MemoryAnalysis [Stack Local]</td>
<td>309,789,752</td>
<td>144</td>
</tr>
<tr>
<td>contextClassLoader</td>
<td>95,953</td>
<td>160</td>
</tr>
<tr>
<td>&lt;local variable&gt; [char[8192] [Stack Local] == (all elements = 0)</td>
<td>16,408</td>
<td>16,408</td>
</tr>
<tr>
<td>&lt;local variable&gt; java.io.BufferedReader [Stack Local]</td>
<td>8,764</td>
<td>48</td>
</tr>
</tbody>
</table>
```

Object class: java.lang.Thread
Object generation: not available
Object index: #1011037
Web application: None
Distance to nearest GC root: 0 (the object is a root itself)

Shallow size: 176
Retained objects (includes the object itself): 7,000,067
Retained size (includes shallow size): 309,816,648
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

» wasting memory – memory doesn’t keep any useful content
  • 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
memory leak – objects are no longer used but there are still references to them
  • 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 – speed of data read / write
  • 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)
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

---

<table>
<thead>
<tr>
<th>Call Tree</th>
<th>Objects</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Objects without allocation information&gt;</td>
<td>92,413</td>
<td>12,760,600</td>
</tr>
<tr>
<td>&lt;All threads&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▼ com.intellij.rt.execution.application.AppMain.main(String[])</td>
<td>25,264</td>
<td>1,692,200</td>
</tr>
<tr>
<td>▼ NativeMethodAccessorImpl.java (native) Lambda.main(String[])</td>
<td>25,264</td>
<td>1,692,200</td>
</tr>
<tr>
<td>▼ Lambda.java:44 Lambda.reversedAlphabeticalOnlyOrder(String[])</td>
<td>25,258</td>
<td>1,667,408</td>
</tr>
<tr>
<td>▼ Lambda.java:13 java.lang.invoke.MethodHandleNatives.linkCallSite(Object, Object, Object, Object, Object[], Object[])</td>
<td>15,709</td>
<td>1,050,728</td>
</tr>
<tr>
<td>▼ Lambda.java:13 java.lang.invoke.MethodHandleNatives.linkMethodHandleConstant(Class, int, Class, String, Object)</td>
<td>5,627</td>
<td>324,184</td>
</tr>
<tr>
<td>▼ Lambda.java:15 java.util.Comparator.comparing(Function)</td>
<td>1,557</td>
<td>106,040</td>
</tr>
<tr>
<td>▼ Lambda.java:16 java.util.stream.Collectors.toList()</td>
<td>1,032</td>
<td>68,912</td>
</tr>
<tr>
<td>▼ Lambda.java:16 java.util.stream.ReferencePipeline.collect(Collectors)</td>
<td>869</td>
<td>71,760</td>
</tr>
<tr>
<td>▼ Lambda.java:14 java.lang.invoke.MethodHandleNatives.linkCallSite(Object, Object, Object, Object, Object[])</td>
<td>161</td>
<td>10,088</td>
</tr>
<tr>
<td>▼ Lambda.java:13 java.lang.invoke.MethodHandleNatives.findMethodHandleType(Class, Class[])</td>
<td>98</td>
<td>5,904</td>
</tr>
<tr>
<td>▼ Lambda.java:13 java.util.Arrays.stream(Object[])</td>
<td>87</td>
<td>17,608</td>
</tr>
<tr>
<td>▼ Lambda.java:16 java.util.stream.Collectors.&lt;cinit&gt;()</td>
<td>33</td>
<td>3,608</td>
</tr>
<tr>
<td>▼ Lambda.java:14 java.lang.invoke.MethodHandleNatives.linkMethodHandleConstant(Class, int, Class, String, Object)</td>
<td>24</td>
<td>872</td>
</tr>
<tr>
<td>▼ Lambda.java:14 java.lang.invoke.MethodHandleNatives.findMethodHandleType(Class, Class[])</td>
<td>18</td>
<td>640</td>
</tr>
</tbody>
</table>

![Call Tree Image]

---

<table>
<thead>
<tr>
<th>Class</th>
<th>Objects</th>
<th>Shallow Size</th>
<th>Retained Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[]</td>
<td>5,280</td>
<td>388,472</td>
<td>388,472</td>
</tr>
<tr>
<td>byte[]</td>
<td>929</td>
<td>284,856</td>
<td>284,856</td>
</tr>
<tr>
<td>jdk.internal.org.objectweb.asm.Item</td>
<td>3,051</td>
<td>170,856</td>
<td>170,856</td>
</tr>
<tr>
<td>java.lang.Class</td>
<td>230</td>
<td>136,328</td>
<td>165,824</td>
</tr>
</tbody>
</table>
Dynamic Memory Analysis – Allocation Tracking

- allocation tracking – flight recording using jmc – no byte code instrumentation
  - identify large object allocations outside TLAB

<table>
<thead>
<tr>
<th>Thread Local Allocation Buffer (TLAB) Statistics</th>
<th>Statistics for Object Allocations (Outside TLABs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLAB Count</td>
<td>Object Count</td>
</tr>
<tr>
<td>393</td>
<td>48</td>
</tr>
<tr>
<td>Maximum TLAB Size</td>
<td>Maximum Object Size</td>
</tr>
<tr>
<td>16.26 MB</td>
<td>16.34 MB</td>
</tr>
<tr>
<td>Minimum TLAB Size</td>
<td>Minimum Object Size</td>
</tr>
<tr>
<td>2.05 kB</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Average TLAB Size</td>
<td>Average Object Size</td>
</tr>
<tr>
<td>2.45 MB</td>
<td>5.67 MB</td>
</tr>
<tr>
<td>Total Memory Allocated for TLABs</td>
<td>Total Memory Allocated for Objects</td>
</tr>
<tr>
<td>961.48 MB</td>
<td>272.12 MB</td>
</tr>
<tr>
<td>Allocation Rate for TLABs</td>
<td>Allocation Rate for Objects</td>
</tr>
<tr>
<td>15.65 MB/s</td>
<td>4.43 MB/s</td>
</tr>
</tbody>
</table>

Allocation

- TLAB Allocations
- Object Allocations (Outside TLABs)
Dynamic Memory Analysis – Allocation Tracking

- allocation tracking – **flight recording** using jmc – no byte code instrumentation
  - identify large object allocations outside TLAB

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

<table>
<thead>
<tr>
<th>Stack Trace</th>
<th>TLABs</th>
<th>Total TLAB Size</th>
<th>Pressure</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.ArrayList.&lt;init&gt;(int)</td>
<td>1</td>
<td>16.26 MB</td>
<td>75.89%</td>
</tr>
<tr>
<td>java.util.AbstractCollection.toArray()</td>
<td>1</td>
<td>1.82 MB</td>
<td>8.49%</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream.defaultWriteFields(Object, ObjectOutputStream)</td>
<td>17</td>
<td>1.32 MB</td>
<td>6.17%</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream$HandleTable$HandleTable$DefaultWriteFields</td>
<td>11</td>
<td>1.27 MB</td>
<td>5.94%</td>
</tr>
<tr>
<td>sun.management.ManagementInstrumentationCompositeData$DefaultWriteFields</td>
<td>4</td>
<td>171.98 kB</td>
<td>0.78%</td>
</tr>
<tr>
<td>sun.reflect.misc.MethodInvocation$DefaultWriteFields</td>
<td>3</td>
<td>164.10 kB</td>
<td>0.75%</td>
</tr>
<tr>
<td>java.io.ObjectStreamClass$DefaultWriteFields$DefaultWriteFields</td>
<td>2</td>
<td>71.89 kB</td>
<td>0.33%</td>
</tr>
<tr>
<td>java.io.ObjectOutputStream$DefaultWriteFields$DefaultWriteFields</td>
<td>1</td>
<td>64.12 kB</td>
<td>0.29%</td>
</tr>
<tr>
<td>java.lang.reflect.Array$DefaultWriteFields$DefaultWriteFields</td>
<td>1</td>
<td>64.05 kB</td>
<td>0.29%</td>
</tr>
<tr>
<td>sun.management.MappedMXBeanType$Map$DefaultWriteFields$DefaultWriteFields</td>
<td>1</td>
<td>52.48 kB</td>
<td>0.24%</td>
</tr>
</tbody>
</table>
Dynamic Memory Analysis – Allocation Tracking

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

- identify large object allocations outside TLAB
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 in the following order (in the same type group respecting declaration order):
    – 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

<table>
<thead>
<tr>
<th>8B - mark word</th>
</tr>
</thead>
<tbody>
<tr>
<td>4B / 8B – Klass ref.</td>
</tr>
<tr>
<td>... object data</td>
</tr>
</tbody>
</table>
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() {...
```
### Data Structures – Object Example 64-bit >=32GB Heap

**Object structure (64-bit JVM) using standard OOP:**

- object size 96 Bytes

<table>
<thead>
<tr>
<th>Address (0x)</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00:</td>
<td>mark word</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00:</td>
<td>Klass ref.</td>
</tr>
<tr>
<td>0x10:</td>
<td>long1</td>
</tr>
<tr>
<td>0x20:</td>
<td>long2</td>
</tr>
<tr>
<td>0x30:</td>
<td>int1</td>
</tr>
<tr>
<td></td>
<td>float1</td>
</tr>
<tr>
<td>0x40:</td>
<td>int2</td>
</tr>
<tr>
<td></td>
<td>float2</td>
</tr>
<tr>
<td>0x50:</td>
<td>char1</td>
</tr>
<tr>
<td></td>
<td>short1</td>
</tr>
<tr>
<td></td>
<td>char2</td>
</tr>
<tr>
<td></td>
<td>short2</td>
</tr>
<tr>
<td></td>
<td>bo1</td>
</tr>
<tr>
<td></td>
<td>by1</td>
</tr>
<tr>
<td></td>
<td>bo2</td>
</tr>
<tr>
<td></td>
<td>by2</td>
</tr>
</tbody>
</table>

**Class Structure:**

```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() {...}
}
```
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**

**Array object structure (64-bit JVM):**
- 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** – 100% efficiency means zero overhead
  \[
  \frac{\text{useful\_content\_size}}{\text{retained\_size}} \times 100\% 
  \]

- 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

<table>
<thead>
<tr>
<th>Object</th>
<th>Useful size</th>
<th>Retained size (Efficiency) &lt;32GB heap</th>
<th>Retained size (Efficiency) &gt;=32GB heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>1 bit</td>
<td>16 B (0.78 %)</td>
<td>24 B (0.52 %)</td>
</tr>
<tr>
<td>Byte</td>
<td>1 B</td>
<td>16 B (6.25 %)</td>
<td>24 B (4.17 %)</td>
</tr>
<tr>
<td>Short, Character</td>
<td>2 B</td>
<td>16 B (12.50 %)</td>
<td>24 B (8.34 %)</td>
</tr>
<tr>
<td>Integer, Float</td>
<td>4 B</td>
<td>16 B (25.00 %)</td>
<td>24 B (16.67 %)</td>
</tr>
<tr>
<td>Long, Double</td>
<td>8 B</td>
<td>24 B (33.34 %)</td>
<td>24 B (33.34 %)</td>
</tr>
</tbody>
</table>
**Primitive Objects**

- **auto boxing** and **un-boxing** during assignment and parameter passing
  - `valueOf({primitive})` and `{primitive}Value()` methods
- All primitive objects are **immutable** (final values)
- Beware of **inefficiencies** caused by boxing and un-boxing

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

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

```
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
```

```
// Field integer:Ljava/lang/Integer;
// Method java/lang/Integer.intValue():II
```
Conversion Inefficiencies - Example

» count word histogram

```java
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

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

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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Object</th>
<th>Useful size</th>
<th>Retained size (Efficiency) &lt;32GB heap</th>
<th>Retained size (Efficiency) &gt;=32GB heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList&lt;Integer&gt;</td>
<td>4 MiB</td>
<td>34.33 MiB (11.65 %)</td>
<td>47.26 MiB (8.46 %)</td>
</tr>
<tr>
<td>ArrayList&lt;Integer&gt;</td>
<td>4 MiB</td>
<td>17.73 MiB (22.56 %)</td>
<td>25.72 MiB (15.55 %)</td>
</tr>
<tr>
<td>HashMap&lt;Integer,Double&gt;</td>
<td>12 MiB</td>
<td>70.19 MiB (17.10 %)</td>
<td>87.67 MiB (13.69 %)</td>
</tr>
</tbody>
</table>
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 approach
  - support big collections (>\(2^{31}\) elements) in *FastUtil*
  - support custom hashing strategies in *Trove*

```java
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
    \[ h(i,k) = (h_1(k) + i \cdot h_2(k)) \mod |T| \]
    – \( h_2 \) 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

**Note:** 1 million of elements stored

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