Intro

- Jan Kotek
  - jan@kotek.net
  - @JanKotek
  - MapDB author (database engine)
  - Java developer for past 12 years
  - Last 3 years independent consultant for MapDB

- Slides and code at
  - https://github.com/jankotek/talk-save-java-memory

Intro

- This talk is about:
  - Reducing memory footprint
  - Object overhead
  - Primitive variables and arrays
  - Alternative Java Collections
- Not about:
  - Garbage Collection
  - Off-heap
  - Performance
  - Memory leaks

Measure memory consumption

- Runtime.getRuntime().freeMemory() difference
- Allocate until OutOfMemoryError
- Profiler
- Class or heap dump analyze such as Java Object Layout

Runtime.getRuntime().freeMemory()

- Get allocated memory stats
- Create new objects
- Call GC many times
- Get allocated memory difference

- Not very precise, but easy
- May not always work,
  some JVMs ignore System.gc()

```
NanoBench nanoBench = NanoBench.create();
nanoBench.memoryOnly().warmUps(2).measurements(10)
.measure(String.format("Measure memory for %d keys", i),
  new Runnable() {
      @Override public void run() {
          Random setRandom = new Random(4532);
          ref = new HashSet<Integer>(cnt);
          w hile (ref.size() < cnt) {
             ref.add(setRandom.nextInt());
          }
      }
    });
nanoBench.getMemoryBytes()
```

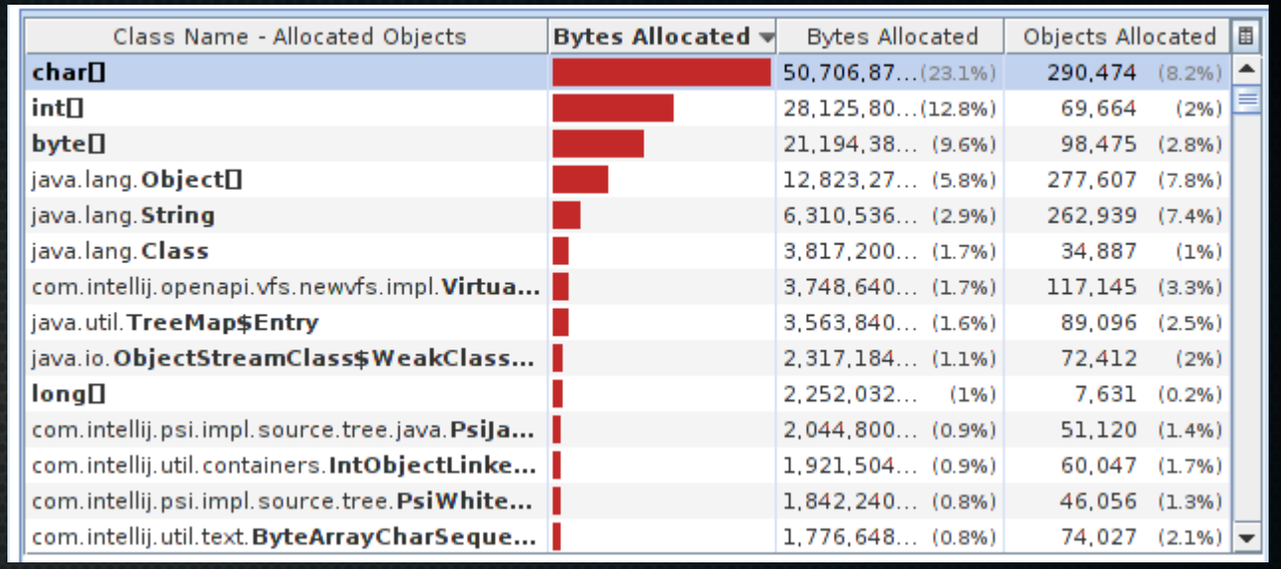
OutOfMemoryError

- Memory limit (-Xmx1G)
- Add entries to array in loop
- Print number of items
- JVM eventually crashes

```
try{
    //add new entry in loop
    while(true)
        array[arrayPos++] = new HashMap();
 }catch(OutOfMemoryError e){
    memoryConsumption =
        Runtime.getRuntime().maxMemory()
            /  counter;
    return;
 }
```

Profiler

- Shows memory consumed by single class
  - (byte[], Object[])
- VisualVM is easy to use
- Flight Recorder is best
  - Free outside production env.



Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		50,706,87... (23.1%)	290,474 (8.2%)
int[]		28,125,80... (12.8%)	69,664 (2%)
byte[]		21,194,38... (9.6%)	98,475 (2.8%)
java.lang.**Object[]**		12,823,27... (5.8%)	277,607 (7.8%)
java.lang.**String**		6,310,536... (2.9%)	262,939 (7.4%)
java.lang.**Class**		3,817,200... (1.7%)	34,887 (1%)
com.intellij.openapi.vfs.newvfs.impl.**Virtua...**		3,748,640... (1.7%)	117,145 (3.3%)
java.util.**TreeMap$Entry**		3,563,840... (1.6%)	89,096 (2.5%)
java.io.**ObjectStreamClass$WeakClass...**		2,317,184... (1.1%)	72,412 (2%)
long[]		2,252,032... (1%)	7,631 (0.2%)
com.intellij.psi.impl.source.tree.java.**PsiJa...**		2,044,800... (0.9%)	51,120 (1.4%)
com.intellij.util.containers.**IntObjectLinke...**		1,921,504... (0.9%)	60,047 (1.7%)
com.intellij.psi.impl.source.tree.**PsiWhite...**		1,842,240... (0.8%)	46,056 (1.3%)
com.intellij.util.text.**ByteArrayCharSeque...**		1,776,648... (0.8%)	74,027 (2.1%)

Java Object Layout

- Command line tool to analyze memory layout (and space usage)
- Most advanced
- http://openjdk.java.net/projects/code-tools/jol/
- Written by Aleksey Shipilev

- Example:

```
$ java -jar jol-cli/target/jol-cli.jar internals java.util.HashMap

Instance size: 48 bytes (reported by Instrumentation API)
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

Instance overhead in Java

- Pointer overhead
  - 4 to 8 bytes
  - depends on JVM and amount of memory
  - 64bit JVM 8 uses compressed pointer when possible
- Class info and object header
  - 12 bytes
- Object alligment padding
  - Between 0 and 15

35 bytes worst case scenario

java.lang.Integer overhead

- One object + one reference
- Total overhead 16 bytes + reference (4 to 8 bytes)

```
JOL output for java.lang.Integer

 OFFSET  SIZE  TYPE DESC

      0    12       (object header)
     12     4   int Integer.value


total size: 16 bytes
```

java.lang.String overhead

- Two objects and two references
- One int field (String.hash)
- Two bytes per char (UTF16)

- Other problems:
  - Weak hash (many collisions)
  - Defensive char[] copies
  - Substring create copies etc..

JOL output for java.lang.String

```
 OFFSET  SIZE   TYPE    DESC
      0     4   (object header)
      4     4   (object header)
      8     4   (object header)
     12     4   char[]  String.value
     16     4   int     String.hash
     20     4   (alignment loss)

total size: 24 bytes
```

String.intern()

- Reduces number of instances
- Each string creates new weak ref (huge GC overhead)
- Long.valueOf(11) is similar

```
name.toLowerCase().intern()
            ==
name.toLowerCase().intern()
```

Represent String as char[]

- Field is char[] or byte[]
- Get/set takes string
- GC overhead, but short lived objects are fast to collect
- Reduced memory with many objects (large Maps etc)

```
public class Person{

    private char[] name;

    public String getName(){
        return new String(name);
    }

    public void setName(String name){
        this.name = name.toCharArray();
    }
}
```

char[] as key in Map

- char[] has no equals() and hashCode() methods
- TreeMap key can use Comparator<char[]>
  - com.google.common.primitives.Chars.lexicographicalComparator()
- java.util.HashMap can not change hash strategy
  - Object2ObjectOpenCustomHashMap from FastUtil has Hash Strategies
  - String.hashCode() is broken, many collisions
    - Part of spec, can not be changed :-(
    - ConcurrentHashMap Java8 has sorted lists to handle hash duplicates :-)

char[] as value in Map

- Works (mostly)
- Concurrent Compare-And-Swap operations broken
  - char[].equals() uses identity
  - ConcurrentMap.replace(K key, V oldValue, V newValue);
  - MapDB Maps allow custom equality

Bitwise

```
class Address{
  String street;
  int houseNumber;
}
```

- Replace street with number from dictionary
- Combine streetId and houseNumber into single long number
  - 5 bytes for streetId, 3 bytes for house number and flags
- Adress become primitive number, no objects or references

Primitive key / value in collections

- HashMap<Long,Long> uses Object[] for keys and values
- ~40 bytes overhead for each key-value pair
- Use long[] or int[] instead

- Data locality → CPU cache friendly
  - Binary search much faster
  - Faster fetch of values
- Almost zero Garbage Collection
  - entire HashMap uses two objects
  - No objects allocated for new key or value

FastUtil

- http://fastutil.di.unimi.it/
- Autogenerated classes with primitive keys and values
- Int2IntMap, Int2BooleanMap, Int2ObjectMap<V> ….
- Many classes, 17MB jar

FastUtil

```
//  create new map
Int2ObjectMap<String> map =
    new Int2ObjectOpenHashMap();

// put/get has primitive arg, no unboxing!
map.put(1, "aa");
map.get(1);

// also implements full mapdb interface
Map<Integer, String> map2 = map;
```

- Parallel interface hierarchy
- Get / set has primitive arg version, no unboxing!
- Implements Map interface

Chronicle Map

- ConcurrentHashMap
- Keys and values are serialized outside heap
- Supports any serializable data type
- No objects or references on heap, memory efficient
- http://chronicle.software/products/chronicle-map/

```
Map<Long, long[]> graph =
ChronicleMap
    .of(Long.class, long[].class)
    .entries(1_000_000_000L)
    .averageValue(new long[150])
    .create();
```

Conclusion

- Avoid too many object instances in long lived objects
- Short lived objects are fine, when traded for smaller memory

- Email: jan@kotek.net

- Slides and code at
  - https://github.com/jankotek/talk-save-java-memory