ORACLE

java.lang.String Catechism

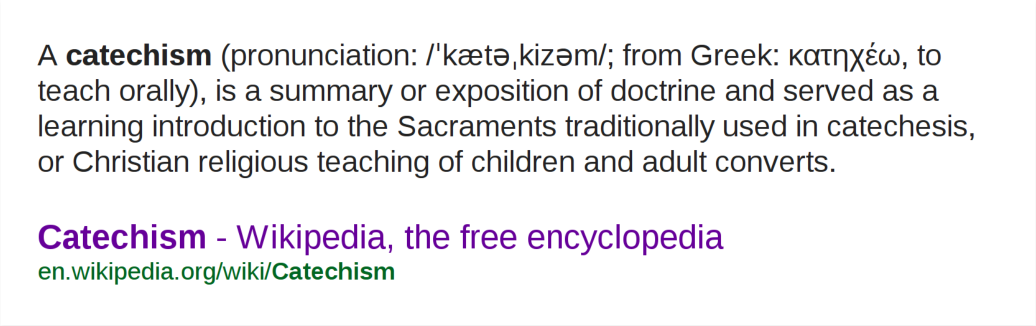
Stay Awhile And Listen

Aleksey Shipilёv

aleksey.shipilev@oracle.com, @shipilev

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Intro



A **catechism** (pronunciation: /ˈkætəˌkizəm/; from Greek: κατηχέω, to teach orally), is a summary or exposition of doctrine and served as a learning introduction to the Sacraments traditionally used in catechesis, or Christian religious teaching of children and adult converts.

Catechism - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/**Catechism**

“Science replaces private prejudice with public, verifiable evidence.”

— Richard Dawkins

Intro: Disclaimers

All tests are done:

- ...by trained professionals: recheck[1] the results before using them
- ...on 1x2x4 i7-4790K (4.0 GHz, HSW): that machine is **fast**
- ...running Linux x86_64, 3.13: latest stable Linux Kernel
- ...with a 8u40 EA x86_64: the latest and greatest JDK
- ...driven by JMH[2]: the latest and greatest benchmarking harness

[1] https://github.com/shipilev/article-string-catechism/

[2] http://openjdk.java.net/projects/code-tools/jmh/

Intro: Strings are abundant

- Humans communicate with text
- Machines follow suit and communicate with text as well: most source code is text, many data interchange formats are text

- Anecdotal data from JEP 192: 25% of heap occupied by String objects
- Anecdotal data: String optimizations usually bring the immediate payoff

Understanding and avoiding cardinal sins is the road to awe.

Internals

Internals: java.lang.String inside

```
public final class String implements ... {
  private final char[] value;
  private int hash;
  ...
```

Strings are immutable:

- Can use/pass them without synchronization, and nothing breaks
- Can share the underlying `char[]` array, covertly from user

Internals: java.lang.String internals

Quite a bit of space overhead:

```
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0    12        (object header)
     12     4 char[] String.value
     16     4    int String.hash
     20     4        (alignment loss)
Instance size: 24 bytes
```

- 8..16 bytes: `String` header
- 4..4 bytes: `String` hashcode
- 12..16 bytes: `char[]` header
- 0..8 bytes: alignment losses

12..24 bytes against `char[]`, 24..44 bytes against `wchar_t*`

Internals: Catechism

Q: Should I use `Strings` to begin with?
A: Absolutely, when you are dealing with text data.

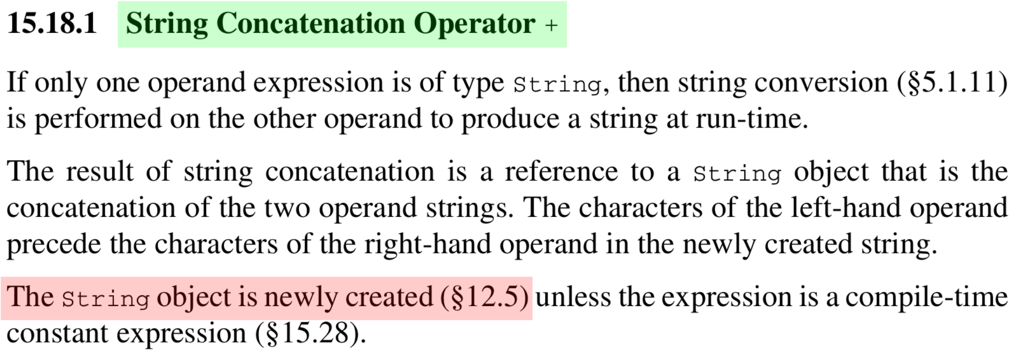
Q: What if memory footprint is a concern?
A: There are remedies for that, read on.

Q: I can wind up my own `String` implementation over `char[]`!
A: Sure you can, read on for caveats.

Q: *Should* I wind up my own `String` implementation?
A: *(Silence was the answer, and Engineer left enlightened)*

Immutable

Immutable: Strings are special



15.18.1 String Concatenation Operator +

If only one operand expression is of type `String`, then string conversion (§5.1.11) is performed on the other operand to produce a string at run-time.

The result of string concatenation is a reference to a `String` object that is the concatenation of the two operand strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string.

The `String` object is newly created (§12.5) unless the expression is a compile-time constant expression (§15.28).

Immutable: Stuck In A Loop

```
@Benchmark
public String string() {
  String s = "Foo";
  for (int c = 0; c < 1000; c++) {
    s += "Bar";
  }
  return s;
}
```

Immutable: Stuck In A Loop

```
@Benchmark
public String string() {
  String s = "Foo";
  for (int c = 0; c < 1000; c++) {
    s += "Bar"; // newly created String here
  }
  return s;
}
```

Immutable: Stuck In A Loop

```
@Benchmark
public String stringBuilder() {
  StringBuilder sb = new StringBuilder();
  for (int c = 0; c < 1000; c++) {
    sb.append("Bar");
  }
  return sb.toString();
}
```

Immutable: Stuck In A Loop

How bad could it be, anyway?

Benchmark	Throughput, ops/s	
string	3250.875	± 18.434
stringBuffer	125270.620	± 1005.263
stringBuilder	116173.291	± 422.926

Lots of pain: here, 30x performance penalty for adding a thousand of Strings.
Compilers are only able to help so much (more later).
My JVM hovercraft is full of GC eels.

Immutable: Catechism

Q: Why this is so painful?
A: Immutability **almost always** comes at a cost.

Q: But I like immutability, how to ease the pain?
A: Use Builders to construct immutable objects.

Q: Why can't JDK/JVM optimize this for us?
A: It can, in many cases. But, there is no escape if you want the best possible performance for all possible cases. (No Free Lunch)

Q: Do I need the best possible performance?
A: *(Silence was the answer, and Engineer left enlightened)*

Concat

Concat: Decompiling

```java
@Benchmark
public String string_2() {
  return s1 + s2;
}
```

...compiles into:

```
public String string_2();
  Code:
     0: new             #14    // java.lang.StringBuilder
     3: dup
     4: invokespecial #15    // StringBuilder.new()
     7: aload_0
     8: getfield        #3     // s1:String;
    11: invokevirtual #16    // StringBuilder.append(String);
    14: aload_0
    15: getfield        #5     // s2:String;
    18: invokevirtual #16    // StringBuilder.append(String);
    21: invokevirtual #17    // StringBuilder.toString();
    24: areturn
```

SB: Decompiling

Not suprisingly,
`StringBuilder.append` chains are routinely optimized:

```
@Benchmark
public String sb_6() {
  return new StringBuilder()
      .append(s1).append(s2).append(s3)
      .append(s4).append(s5).append(s6)
      .toString();
}

@Benchmark
public String string_6() {
  return s1 + s2 + s3 + s4 + s5 + s6;
}
```

Try this with `-XX:±OptimizeStringConcat` to quantify...

SB: StringBuilder opts are good!

Benchmark	N	Score, ns/op				Impr
		-Opt		+Opt		
stringBuilder	1	13.993	± 0.079	8.694	± 0.080	+61%
stringBuilder	2	20.259	± 0.181	12.042	± 0.370	+68%
stringBuilder	3	27.015	± 0.224	14.831	± 0.068	+82%
stringBuilder	4	33.344	± 0.546	21.068	± 0.087	+58%
stringBuilder	5	38.151	± 0.216	25.454	± 0.122	+50%
stringBuilder	6	69.626	± 1.042	29.856	± 0.221	+133%
string	1	2.273	± 0.013	2.273	± 0.004	0%
string	2	20.410	± 0.150	11.793	± 0.055	+73%
string	3	27.059	± 0.311	14.897	± 0.075	+82%
string	4	32.952	± 0.446	21.122	± 0.177	+56%
string	5	37.978	± 0.321	25.349	± 0.141	+50%
string	6	70.134	± 0.728	29.895	± 0.334	+135%

SB: Implicit SB vs. Explicit Conversion

Because of that, people are surprised how this benchmark behaves:

```
private int x;

@Setup
void setup() { x = 1709; }

@Benchmark
String concat_Pre()      { return "" + x; }

@Benchmark
String concat_Post()     { return x + ""; }

@Benchmark
String integerToString() { return Integer.toString(x); }

@Benchmark
String stringValueOf()   { return String.valueOf(x); }
```

SB: Implicit SB vs. Explicit Conversion (cont.)

Benchmark	Score, ns/op	
concat_Post	14.962	± 0.136
concat_Pre	15.063	± 0.198
integerToString	21.824	± 0.181
stringValueOf	21.979	± 0.312

Implicit concatenation is faster than **explicit** conversions?

- `StringBuilder` optimizations kick in, and `append(int)` is actually faster!
- And will be slower with `-XX:-OptimizeStringConcat`

SB: Side Effects

Let’s make it a little bit more complicated...

```
private int x;

@Setup
void setup() { x = 1709; }

@Benchmark
String concat_just()          { return "" + x; }

@Benchmark
String concat_side()          { x--; return "" + (x++); }

@Benchmark
String integerToString_just() { return Integer.toString(x); }

@Benchmark
String integerToString_side() { x--; return Integer.toString(x++); }
```

SB: Side Effects (cont.)

Benchmark	Score, ns/op	
concat_just	14.868	± 0.057
integerToString_just	21.684	± 0.094
stringValueOf_just	21.622	± 0.090
concat_side	27.263	± 0.262
integerToString_side	21.625	± 0.093
stringValueOf_side	21.682	± 0.138

- Once we have a side-effect in `append()` call, optimization bails out[3]
- On deopt, need to «unwind» the execution, but unable to do so for stores
- Moving the memory stores out of `append()` args helps

[3] https://bugs.openjdk.java.net/browse/JDK-8043677

Lazy Logging: Trouble

```java
private int x;
private boolean enabled;

void log(String msg) {
  if (enabled) {
    System.out.println(msg);
  }
}

@Benchmark
void heap_string() {
  log("Wow,␣x␣is␣such␣" + x + "!");
}

@Benchmark
void heap_string_guarded() {
  if (enabled) {
    log("Wow,␣x␣is␣such␣" + x + "!");
  }
}
```

- Concatenation happens before the `enabled` check
- Wasting precious time constructing the strings we don't care about
- Therefore, most people opt to guard the logger calls before even touching the strings

Lazy Logging: Trouble

```
private int x;
private boolean enabled;

@Benchmark
void heap_lambda() {
  log(() -> "Wow,␣such␣" + x + "!");
}

@Benchmark
void noArg_lambda() {
  log(() -> "Such␣message,␣wow.");
}

@Benchmark
public void local_lambda() {
  int lx = x;
  log(() -> "Wow,␣such␣" + lx + "!");
}
```

- We can do much better with lambdas: deferred execution without a syntactic mess
- There is a bit of the underlying difference when referencing locals, fields, or nothing

Lazy Logging: Lazy Logging

Method	Time, ns/op		
	heap	local	noArgs
string	19.298 ± 0.399	17.718 ± 0.248	0.381 ± 0.007
lambda	1.893 ± 0.011	1.809 ± 0.019	0.385 ± 0.013
string_guarded	0.385 ± 0.010	0.381 ± 0.004	0.383 ± 0.007

Lambdas rock! The explicit guard still wins, but not by a large margin: capturing lambdas (yet) need instantiation.

Concat: Catechism

Q: Should I be worried about concatenation costs?
A: You should in all non-trivial cases. You can't help much in trivial cases.

Q: What concatenation cases are non-trivial?
A: Any pattern involving control flow, side effects, unpredictable values.

Q: Are `StringBuilder`-s flawless?
A: They are aggressively optimized, but sometimes even those optos fail.

Q: I am PL professional, give me lazy-val, call-by-name, and shut up.
A: *(points to JDK 8 release, and PL professional leaves enlightened)*

Hash Codes

Zeroes: P(31) hashcode

Spec says `String.hashCode` is a P(31) polynomial hashcode:

$$h(s) = \sum_{k=0}^{n-1} 31^{n-k-1} s_k$$

```
public int hashCode() {
  ...
  int h = 0;
  for (char v : value) {
    h = 31 * h + v;
  }
  hash = h;
}
```

Time complexity is $\Omega(N)$ and $O(N)$.

Zeroes: Trying...

```
String str1, str2;

@Setup
public void setup() {
  str1 = "лжеотождествление␣электровиолончели"; // same length
  str2 = "электровиолончели␣лжеотождествление"; // same length
}

@Benchmark
int test1() { return str1.hashCode(); }

@Benchmark
int test2() { return str2.hashCode(); }
```

Zeroes: Trying...

```
String str1, str2;

@Setup
public void setup() {
  str1 = "лжеотождествление␣электровиолончели"; // same length
  str2 = "электровиолончели␣лжеотождествление"; // same length
}

@Benchmark
int test1() { return str1.hashCode(); } // 22.663 ± 0.056 ns/op

@Benchmark
int test2() { return str2.hashCode(); } // 0.758 ± 0.002 ns/op
```

Zeroes: Actual Implementation

```
public int hashCode() {
  int h = hash;
  if (h == 0) {
    for (char v : value) {
      h = 31 * h + v;
    }
    hash = h;
  }
  return h;
}
```

- Actual code caches hashcodes
- Immense improvements in most scenarios, justifying 4 bytes per instance
- By *pigeonhole principle*, some Strings are bound to have $hs(s) = 0$, sucks to be them
- It is a sane engineering tradeoff to have a performance anomaly with 2^{-32} probability

Collisions: Walking on a Sunshine

```
// carefully populated with unicorn dust:
HashMap<String, String> sunshine;

@Benchmark void keySet(Blackhole bh) {
  for (String key : sunshine.keySet()) {
    bh.consume(sunshine.get(key));
  }
}

@Benchmark void entrySet(Blackhole bh) {
  for (Map.Entry<String, String> e : sunshine.entrySet()) {
    bh.consume(e);
  }
}
```

Collisions: Using JDK 7u0...

Benchmark	Size	Time, ns/op		ns/key
entrySet	1	14.134	± 0.028	14.1
entrySet	10	47.427	± 0.269	4.7
entrySet	100	294.148	± 0.934	2.9
entrySet	1000	5366.982	± 802.857	5.4
entrySet	10000	67394.472	± 456.576	6.7
keySet	1	18.463	± 0.500	18.4
keySet	10	279.816	± 6.783	27.8
keySet	100	22266.667	± 179.695	222.7
keySet	1000	2716486.481	± 10145.741	2716.5
keySet	10000	355309390.210	± 1214802.832	355309.4

keySet performance rapidly deteriorates: $O(N^2)$

Collisions: Algorithmic Attacks

Polynomial hash functions make artificial collisions a piece of cake.
Suppose this expansion:

$$h(s) = \sum_{k=0}^{n-1} 31^{n-k-1} s_k = [\sum_{k=0}^{n-3} 31^{n-k-1} s_k] + 31 s_{n-2} + s_{n-1}$$

Then, if strings a and b have common prefix in $[0..n-3]$:

$$h(a) = h(b) \Leftrightarrow 31(a_{n-2} - b_{n-2}) = (a_{n-1} - b_{n-1})$$

...and that is super-easy, suppose $a = $"$...Aa$" and $b = $"$...BB$".

Collisions: Why should I care?

- Alice is running her battle-hardened HTTP server, patched up for Heartbleed, Shellshock, all these fancy-named vulnerabilities.
  Alice is serious about security.
- Mallory giggles and sends the HTTP Request with these HTTP Headers:

```
"X-Conference-AaAaAaAa:␣JokerConf␣2014,␣Why␣So␣Serious?"
"X-Conference-AaAaAaBB:␣JokerConf␣2014,␣Why␣So␣Serious?"
"X-Conference-AaAaBBAa:␣JokerConf␣2014,␣Why␣So␣Serious?"
"X-Conference-AaAaBBBB:␣JokerConf␣2014,␣Why␣So␣Serious?"
```

- Alices's web server accepts the request, stores HTTP Headers in `Map<String, String>`, and then tries to process them. Boom, resource exhaustion and possible DoS.

Collisions: Using JDK 8

Benchmark	Size	Time, ns/op		ns/key
entrySet	1	11.674	± 0.040	11.7
entrySet	10	36.301	± 0.076	3.6
entrySet	100	278.057	± 0.726	2.8
entrySet	1000	3606.722	± 21.441	3.6
entrySet	10000	86459.477	± 626.407	8.6
keySet	1	15.050	± 0.084	15.0
keySet	10	253.241	± 0.650	2.5
keySet	100	10072.577	± 144.418	100.7
keySet	1000	158591.766	± 1202.430	158.6
keySet	10000	2355039.389	± 12087.352	235.3

keySet is now $O(NlogN)$ – not as bad

Collisions: Another quirks

`http://www.zlib.net/crc_v3.txt`

> *In particular, any CRC algorithm that initializes its register to zero will have a blind spot of zero when it starts up and will be unable to "count"a leading run of zero bytes. As a leading run of zero bytes is quite common in real messages, it is wise to initialize the algorithm register to a non-zero value.*

The same applies to `String.hashCode`.
Thank God, NUL-prefixed Strings are not common.

Hash Codes: Catechism

Q: Should I care about `String.hashCode`?
A: Most likely not, unless you expose your naked `Maps` for user input.

Q: Should I wrap the `Strings` with my own `hashCode` implementation?
A: In some very rare cases, yes.

Q: Why TIAO wouldn't change the `String.hashCode` computation?
A: The P(31) hashcode is spec-ed in so many places, it can't be changed now.

Q: That `hashCode` caching thing at zero bothers me, can be do a `boolean` flag?
A: That will explode `String` footprint by 8 bytes in worst case.

Substring

Substring: JDK 8

```
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0    12        (object header)
     12     4 char[] String.value
     16     4    int String.hash
     20     4        (alignment loss)
Instance size: 24 bytes
```

Seasoned Java devs can wonder...

Substring: JDK 8

```
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0    12        (object header)
     12     4 char[] String.value
     16     4    int String.hash
     20     4        (alignment loss)
Instance size: 24 bytes
```

Seasoned Java devs can wonder... where are `offset` and `count` fields?

Substring: JDK < 7u6

```
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION
      0    12        (object header)
     12     4 char[] String.value
     16     4    int String.offset
     20     4    int String.count
     24     4    int String.hash
     28     4        (alignment loss)
Instance size: 32 bytes
```

Here they are! Left behind the enemy lines in JDK 7.

Substring: Benchmark

```
@Param({"0", "30", "60", "90", "120"})
int limit;

String str;

@Setup
public void setup() {
  str = "JokerConf␣2014:␣Why␣So␣Serious?␣" +
        "JokerConf␣2014:␣Why␣So␣Serious?␣" +
        "JokerConf␣2014:␣Why␣So␣Serious?␣" +
        "JokerConf␣2014:␣Why␣So␣Serious?␣";
}

@Benchmark
String head() { return str.substring(limit); }

@Benchmark
String tail() { return str.substring(0, limit); }
```

Substring: JDK $<$ 7u6: Sharing

Limit	Score, ns/op			
	head		tail	
0	2.278	± 0.007	3.763	± 1.091
30	3.566	± 0.261	3.626	± 0.787
60	3.524	± 0.159	3.466	± 0.188
90	3.763	± 0.431	3.464	± 0.089
120	3.713	± 1.053	3.446	± 0.141

- `substring()` only instantiates `Strings`, shares `char[]` arrays
- This is believed to cause memory leaks: think large XML and substring on it

Substring: JDK 8: Copying

Limit	Score, ns/op			
	head		tail	
0	2.277	± 0.012	19.401	± 0.317
30	22.976	± 0.074	10.066	± 0.049
60	16.875	± 0.071	15.202	± 0.116
90	12.782	± 0.088	21.720	± 0.574
120	11.086	± 0.354	26.602	± 0.123

- `substring()` now copies the entire `char[]` array
- Works reasonably well for small substrings, avoids memory leaks

Substring: Catechism

Q: New `substring` sounds bad, can I get it back?
A: No, you can't.

Q: But why?
A: Real memory leaks are worse than potential performance issues.

Q: What if I need $O(1)$ `substring`?
A: That means you care about this enough to make your own storage.

Q: But my application was using `substring` for performance reasons!
A: *(Points to a String.substring Javadoc, and Engineer leaves enlightened)*

Intern

Intern: Interning vs. Deduplication

Deduplication:
Reduce # of instances in each equivalence class

Interning (canonicalization):
Reduce # of instances in each equivalence class to one (canonical) instance.

- As usual, enforcing *stronger* property costs more
- In many cases, you want *deduplication*, not *interning*

Intern: User Interners

Interning is dead-simple, and can be done by hand:

```
public class CHMInterner<T> {
  private final Map<T, T> map;

  public CHMInterner() {
    map = new ConcurrentHashMap<>();
  }

  public T intern(T t) {
    T exist = map.putIfAbsent(t, t);
    return (exist == null) ? t : exist;
  }
}
```

Intern: User Interners (cont.)

Benchmark	Size	Time, us/op	
chm	100	2.448	± 0.014
chm	10000	242.994	± 0.944
chm	1000000	47537.076	± 2123.834
hm	100	0.929	± 0.008
hm	10000	133.796	± 0.748
hm	1000000	35349.188	± 1188.810
intern	100	8.011	± 0.277
intern	10000	891.871	± 13.602
intern	1000000	315664.776	± 17821.360

(Throw-away) `(Concurrent)HashMap` is order of magnitude better!

Intern: And the reason is:

`String.intern()` is a gateway to VM internal StringTable.
StringTable is fixed-size, and almost always overloaded:

```
-XX:+PrintStringTableStatistics
StringTable statistics:
Number of buckets       :      60013 =     480104 bytes, avg    8.000
Number of entries       :    1002451 =   24058824 bytes, avg   24.000
Number of literals      :    1002451 =   64168512 bytes, avg   64.012
Total footprint         :            =   88707440 bytes
Average bucket size     :     16.704
Variance of bucket size :      9.731
Std. dev. of bucket size:      3.119
Maximum bucket size     :         27
```

User-issued `String.intern()` calls only make it worse!

Intern: User Deduplicators

Relaxing the canonicalization requirement may bring the performance:

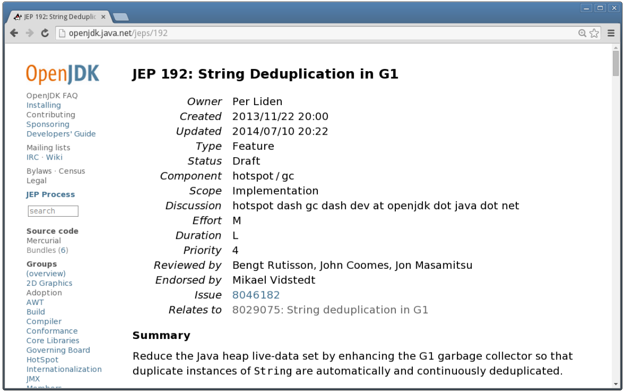
```java
public class CHMDeduplicator<T> {
  private final int prob;
  private final Map<T, T> map;

  public CHMDeduplicator(double prob) {
    this.prob = (int) (Integer.MIN_VALUE + prob * (1L << 32));
    this.map = new ConcurrentHashMap<>();
  }

  public T dedup(T t) {
    if (ThreadLocalRandom.current().nextInt() > prob) {
      return t;
    }
    T exist = map.putIfAbsent(t, t);
    return (exist == null) ? t : exist;
  }
}
```

Intern: Probabilistic Deduplicators

	time, us/op					
Prob	chm		hm		intern	
0.0	3.291	± 0.039	3.286	± 0.030	3.336	± 0.084
0.1	6.953	± 0.039	7.289	± 0.760	13.165	± 0.109
0.2	10.437	± 0.348	9.723	± 0.669	22.493	± 0.127
0.3	13.416	± 0.156	12.027	± 0.146	31.983	± 0.257
0.4	16.457	± 0.098	14.162	± 0.081	40.367	± 0.292
0.5	19.146	± 0.123	15.926	± 0.141	49.379	± 0.806
0.6	21.727	± 1.049	16.693	± 0.285	56.614	± 0.595
0.7	22.465	± 0.154	15.996	± 0.135	63.389	± 1.061
0.8	23.712	± 0.568	15.414	± 0.092	70.731	± 2.515
0.9	25.775	± 0.961	13.986	± 0.121	76.481	± 0.770
1.0	26.140	± 0.089	11.582	± 0.046	118.165	± 30.009

Intern: GC Deduplication

Why can't JVM do this for us?



JEP 192: String Deduplication in G1

Owner	Per Liden
Created	2013/11/22 20:00
Updated	2014/07/10 20:22
Type	Feature
Status	Draft
Component	hotspot / gc
Scope	Implementation
Discussion	hotspot dash gc dash dev at openjdk dot java dot net
Effort	M
Duration	L
Priority	4
Reviewed by	Bengt Rutisson, John Coomes, Jon Masamitsu
Endorsed by	Mikael Vidstedt
Issue	8046182
Relates to	8029075: String deduplication in G1

Summary

Reduce the Java heap live-data set by enhancing the G1 garbage collector so that duplicate instances of `String` are automatically and continuously deduplicated.

```
-XX:+UseG1GC -XX:+UseStringDeduplication
```

Intern: GC Deduplication

```
public static void main(String... args) {
  List<String> strs = ...;

  String last = GraphLayout.parseInstance(strs).toFootprint();
  System.out.println("***␣Original:␣" + last);

  for (int gc = 0; gc < 100; gc++) {
    String cur = GraphLayout.parseInstance(strs).toFootprint();

    if (!cur.equals(last)) {
      System.out.println("***␣GC␣changed:␣" + cur);
      last = cur;
    }

    System.gc();
  }
}
```

Use JOL[4] to estimate the memory footprint.

[4] http://openjdk.java.net/projects/code-tools/jol/

Intern: GC Deduplication

```
*** Original:
java.util.ArrayList instance footprint:
     COUNT       AVG       SUM   DESCRIPTION
     10000        47    472000   [C
         1     56232     56232   [Ljava.lang.Object;
     10000        24    240000   java.lang.String
         1        24        24   java.util.ArrayList
     20002              768256   (total)

*** GC changed:
java.util.ArrayList instance footprint:
     COUNT       AVG       SUM   DESCRIPTION
       100        47      4720   [C
         1     56232     56232   [Ljava.lang.Object;
     10000        24    240000   java.lang.String
         1        24        24   java.util.ArrayList
     10102              300976   (total)
```

Notice the `char[]` arrays are de-duplicated.

Intern: GC Deduplication

```
*** GC changed:
java.util.ArrayList instance footprint:
     COUNT          AVG         SUM   DESCRIPTION
       100           47        4720   [C
         1        56232       56232   [Ljava.lang.Object;
     10000           24      240000   java.lang.String
         1           24          24   java.util.ArrayList
     10102                   300976   (total)

*** Dedup:
java.util.ArrayList instance footprint:
     COUNT          AVG         SUM   DESCRIPTION
       100           47        4720   [C
         1        56232       56232   [Ljava.lang.Object;
       100           24        2400   java.lang.String
         1           24          24   java.util.ArrayList
       202                    63376   (total)
```

Hand-rolled deduplicator can also reduce the number of `String`-s.

Intern: Catechism

Q: But I read so much on using `String.intern` for improving footprint.
A: `http://en.wikipedia.org/wiki/Hanlon's_razor`

Q: I will use `String.intern` just on this tiny little location.
A: Excellent, you already know where your bottlenecks are going to be.

Q: Why wouldn't TIAO optimize `String.intern`?
A: We **are** improving it. It does not help the *misuse* of `String.intern`.

Q: Should I rely on GC deduplication for ultimate memory savings?
A: Identity rules disallow us to merge objects, you have to merge them yourself.

Equals

Equals: Testing basic things

```
String bar10_0 = "BarBarBarA", bar10_1 = "BarBarBarA";
String bar10_2 = "BarBarBarB", bar10_3 = "ABarBarBar";
String bar11   = "BarBarBarAB";

@Benchmark
boolean sameChar()          { return bar10_0.equals(bar10_1); }

@Benchmark
boolean sameLen_diffEnd()   { return bar10_0.equals(bar10_2); }

@Benchmark
boolean sameLen_diffStart() { return bar10_0.equals(bar10_3); }

@Benchmark
boolean differentLen()      { return bar10_0.equals(bar11); }
```

Equals: Basic characteristics

Benchmark	Score, ns/op	
sameChar	0.994	± 0.044
differentLen	1.316	± 0.007
sameLen_diffEnd	4.556	± 0.014
sameLen_diffStart	2.565	± 0.010

- Strings instantiated off the same constant are interned, == check is fast
- Strings of different lengths are not compared at all
- Strings are matched from start to end

Equals: Implementation

```
public boolean equals(Object anObject) {
  if (this == anObject) {
    return true;
  }
  if (anObject instanceof String) {
    String anotherString = (String)anObject;
    int n = value.length;
    if (n == anotherString.value.length) {
      char v1[] = value;
      char v2[] = anotherString.value;
      int i = 0;
      while (n-- != 0) {
        if (v1[i] != v2[i])
          return false;
         i++;
      }
      return true;
    }
  }
  return false;
}
```

«I think this version is well-optimized, and you can gain nothing here...»
(somebody on StackOverflow)

Equals: Intrinsics

Benchmark	Score, ns/op			
	default		disabled[5]	
sameChar	0.994	± 0.044	1.011	± 0.003
differentLen	1.316	± 0.007	1.325	± 0.015
sameLen_diffEnd	4.556	± 0.014	9.654	± 0.052
sameLen_diffStart	2.565	± 0.010	2.989	± 0.050

- The actual `equals()` implementation is intrinsified
- Blindly rewriting the Java implementation will not be faster
- How can intrinsified implementation be 2x faster than «optimal» Java code?

[5] `-XX:+UnlockDiagnosticVMOptions -XX:DisableIntrinsic=::_equals`

Equals: Intrinsics (cont.)

Intrinsic version is vectorized:

```
  5.23%     3.42%    0x00007f1b8c93de95: mov    (%rdi,%rcx,1),%ebx
 14.73%     4.01%    0x00007f1b8c93de98: cmp    (%rsi,%rcx,1),%ebx
                     0x00007f1b8c93de9b: jne    0x00007f1b8c93debb
 26.39%    27.41%    0x00007f1b8c93de9d: add    $0x4,%rcx
                     0x00007f1b8c93dea1: jne    0x00007f1b8c93de95
```

- Notice comparing in 4-byte strides
- This works regardless of whether compiler can or can't auto-vectorize
- VM will select SSE, AVX, etc to efficiently compare.

Equals: Catechism

Q: I have this very nifty idea of optimizing `String.equals`...
A: If you are not prepared to deal with low-level assembly, do not even start.

Q: Why would you need a Java version for `String.equals` then?
A: Interpreter, C1, and other compilers still use this as the fallback code.

Q: Should I intern the `Strings` and then == on them instead?
A: It would be easier to just check the `hashCode` before.

Q: But interning is so much easier!
A: *(silence is the answer, and Programmer leaves enlightened)*

Regexps

Regexps: splitting

```
String text = "Глокая␣куздра␣штеко␣будланула␣бокра␣и␣курдячит␣бокрёнка
String textDup = text.replaceAll("␣", "␣␣");
Pattern pattern = Pattern.compile("␣␣");

@Benchmark
String[] charSplit()          { return text.split("␣"); }

@Benchmark
String[] strSplit()           { return textDup.split("␣␣"); }

@Benchmark
String[] strSplit_pattern() { return pattern.split(textDup); }
```

Regexps: Splitting

Benchmark	Time, ns/op	
charSplit	191.657	± 1.798
strSplit	527.952	± 5.578
strSplit_pattern	416.219	± 4.075

- charSplit has a fast-path for a single-char patterns
- strSplit uses Pattern to match: do not be suprised it works much slower
- strSplit_pattern reuses the Pattern: saves a few cycles

Regexps: Other methods

Lots of other `String` methods are using `Pattern` implicitly:

- `matches(String regex)`
- `replaceFirst(String regex, String replacement)`
- `replaceAll(String regex, String replacement)`
- `replace(CharSequence target, CharSequence replacement)`
- `split(String regex)`
- `split(String regex, int limit)`

You may want to cache `Pattern` in performance-critical places.

Regexps: Backtracking

Searching with `Pattern.compile("(x+x+)+y")`:

Text	Time, ns/op		
size	"xx...xxy"		"xx..xx"
4	94.520	± 1.270	
6	96.848	± 0.936	
8	102.765	± 1.568	
10	106.553	± 5.027	
12	106.786	± 1.515	
14	111.983	± 1.573	
16	115.642	± 2.114	

Regexps: Backtracking

Searching with `Pattern.compile("(x+x+)+y")`:

Text	Time, ns/op			
size	"xx...xxy"		"xx..xx"	
4	94.520	± 1.270	291.830	± 9.274
6	96.848	± 0.936	1049.571	± 7.291
8	102.765	± 1.568	4028.029	± 49.917
10	106.553	± 5.027	15900.084	± 263.320
12	106.786	± 1.515	61694.528	± 704.420
14	111.983	± 1.573	245397.200	± 1528.407
16	115.642	± 2.114	989130.322	± 11201.690

Given the mismatching text, the regexp catastrophically backtracks.

Regexps: Catechism

Q: Should I care? I would never use regular expressions.
A: Yes, you will. Learn how to deal with them before it's too late.

Q: Okay, what are the major improvements I can do?
A: Simplify and cache `Pattern`-s.

Q: Catastrophic backtracking sounds very theoretical, do I have to care?
A: Yes. Unsanitized texts and/or unsanitized regexps are the way to DoS.

Q: Stand back! I know Regular Expressions!
A: *(stands back, and Engineer smacks into wall achieving enlightenment.)*

Walking

Walking: charAt vs toCharArray

```
@Benchmark
public int charAt() {
  int r = 0;
  for (int c = 0; c < text.length(); c++) {
    r += text.charAt(c);
  }
  return r;
}

@Benchmark
public int toCharArray() {
  int r = 0;
  char[] chars = text.toCharArray();
  for (int c = 0; c < text.length(); c++) {
    r += chars[c];
  }
  return r;
}
```

Walking: charAt vs toCharArray

Benchmark	Size	Time, ns/op
charAt	1	2.152 ± 0.002
charAt	10	4.794 ± 0.001
charAt	100	51.579 ± 0.016
charAt	1000	734.582 ± 0.335
toCharArray	1	6.502 ± 0.034
toCharArray	10	9.951 ± 0.050
toCharArray	100	61.204 ± 1.179
toCharArray	1000	1242.236 ± 4.591

- `charAt` bound-checks, but those are nicely optimized out
- `toCharArray` pays for spare memory allocation

Walking: charAt vs toCharArray (spoiled)

```
@Benchmark
public int charAt_spoil() {
  int r = 0;
  for (int c = 0; c < text.length(); c++) {
    spoiler(); // empty non-inlineable
    r += text.charAt(c);
  }
  return r;
}

@Benchmark
public int toCharArray_spoil() {
  int r = 0;
  char[] chars = text.toCharArray();
  for (char c : chars) {
    spoiler(); // empty non-inlineable
    r += c;
  }
  return r;
}
```

Walking: charAt vs toCharArray (spoiled)

Benchmark	size	Score, ns/op	
charAt_spoil	1	4.750	± 1.073
charAt_spoil	10	32.306	± 0.019
charAt_spoil	100	607.965	± 0.206
charAt_spoil	1000	10247.538	± 1552.360
toCharArray_spoil	1	8.903	± 0.042
toCharArray_spoil	10	28.550	± 0.100
toCharArray_spoil	100	435.444	± 3.398
toCharArray_spoil	1000	6559.925	± 22.723

- When VM is unable to track `text`, devirt and bounds-check elimination fail
- Local array is perfectly fine

Walking: Catechism

Q: Should I copy out the `char[]` array or not?
A: If you don't need performance, both approaches are the question of style.

Q: I care about performance, should I copy out the `char[]` array?
A: You should, in non-trivial case.

Q: What is considered non-trivial case?
A: Non-local control flow, volatile reads, etc. that break commonning.

Q: This sucks. There is no universal best-performance way?
A: *(silence was the answer, and Engineer left enligthened)*

Search

Search: Character searches

Searching in `"abcdefghiklmnopqrstuvxyz"`:

image	Time, ns/op			
	indexOf		lastIndexOf	
a	1.306	± 0.001	8.557	± 0.036
m	4.879	± 0.002	5.738	± 0.006
z	7.360	± 0.010	1.677	± 0.000

- Both `indexOf` and `lastIndexOf` are $O(n)$, obviously
- Either is more performant if searched from the start or the end

Search: Intrinsics

Benchmark	Image	Score, ns/op			
		+Opt		-Opt[6]	
indexOf	abc	5.036	± 0.002	4.912	± 0.080
indexOf	mno	7.049	± 0.004	9.875	± 0.076
indexOf	xyz	11.595	± 0.004	12.790	± 0.071
lastIndexOf	abc	13.977	± 0.034	13.956	± 0.031
lastIndexOf	mno	10.588	± 0.003	10.588	± 0.004
lastIndexOf	xyz	5.369	± 0.002	5.370	± 0.002

- Real implementation of `indexOf` is intrinsified
- Uses SSE/AVX extensions to search for a match

[6] `-XX:+UnlockDiagnosticVMOptions -XX:DisableIntrinsic=::_indexOf`

Search: Genome Search

Searching for a sequence of codons in Human Y chromosome:

Benchmark	Time, ms/op	
indexOf	48.262	± 0.434
wikipediaBM	16.741	± 0.497

- `str.indexOf(im)` is a naive search
- `wikipediaBM` is the copy-paste from Boyer-Moore wiki page[7]

[7] `http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm`

Search: Genome Search

Searching for a sequence of codons in Human Y chromosome:

Benchmark	Time, ms/op	
indexOf	48.262	± 0.434
wikipediaBM	16.741	± 0.497
matcherFind	21.223	± 0.429

- `str.indexOf(im)` is a naive search
- `wikipediaBM` is the copy-paste from Boyer-Moore wiki page[7]
- `pattern(im).matcher(str).find()` also uses BM

[7] http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

Search: Catechism

Q: Why there is no optimal string search algo in JDK?
A: «Optimal» is in the eye of beholder.

Q: Why would you maintain a trivial `String.indexOf` anyway?
A: Small images are working better with trivial search.

Q: Java sucks for <insert domain here> because of `indexOf`.
A: *(points to 3rd party libraries, and Engineer leaves enlightened)*

Conclusion

Conclusion: ...

- `String`s are well-optimized:
  - Learning what optimizations are there, and how you can employ them is a useful skill
  - Learning what JDK/VM does is a useful skill
- Performance advice has a generally low «shelf life»:
  - Re-learn stuff as you go
  - Do not trust folklore