Base Language - Feature #6820

reduce String.toLowerCase and toUpperCase usage and String instances

10/06/2022 02:51 AM - Constantin Asofiei

Status:	Review	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Hynek Cihlar	**% Done:**	100%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

History

#1 - 10/06/2022 03:08 AM - Constantin Asofiei

- Subject changed from reduce String.toLowerCase and toUpperCase usage to reduce String.toLowerCase and toUpperCase usage and String instances

4GL is by default case-insensitive. This proves tricky in FWD, as we need to i.e. keep registries with lowercased strings, and even in Text.value , the value is lowercased in case of comparison for case-insensitive character.

Doing a lowercase or uppercase always creates a new String. With 6129a, there are 10-20 million String instances being created just for some complex tests. YourKit allows memory analysis to track object allocation - but even the profiler allows you to see <init> constructor calls, and track down from where is being called (the 'backtraces').

Combined, toLowerCase and toUpperCase is being called some 6 million times for the tests ran with 6129a.

In some cases, it may be possible to keep beside the 'real' value the lowercased value (I'm thinking of case-insensitive character/longchar instances). Having this lowercased version cached, it allows you to re-use it, instead of calling toLowerCase each and every time.

In 3821c and 6129a there will be some improvements to reduce this, and now the highest allocators of String instances are AnnotatedAst.getPath and searchUp, even if a StringBuilder is used. #6813 is meant to improve this.

#2 - 10/06/2022 04:26 AM - Constantin Asofiei

Some changes are in 3821c/14272. Greg, please review.

#3 - 10/10/2022 05:30 AM - Constantin Asofiei

New changes are in 3821c/14281:

- Reduced 'toLowerCase' usage in 'BufferImpl.attachDataSource'.

#4 - 10/26/2022 08:47 AM - Constantin Asofiei

String.toLowerCase usage in a large application (2.1 million calls):

- DmoMeta.byLegacyName - 590k
- AnnotatedAst.isAnnotation - 216k
- BufferImpl.attachDataSource - 127k
- annotatedAst.putAnnotationImpl - 98k
- P2JField.computeHash - 90k
- DataSet$Builder.addDataRelation - 88k
- TempTableBuilder.addAllFields - 84k
- BufferImpl.fill - 80k
- BufferManager.deregisterDynamicBuffer - 60k
- ConnectionManager.getLDBName - 54k
- TemporaryBuffer.createPropsMap@ - 54k
- BufferManager.registerBuffer - 48k
- BufferManager$TempTableKey.<init> - 40k
- SharedVariableManager.addWorker - 32k
- SourceNameMapper$InternalEntryKey.<init> - 30k
- TransactionManager$TransactionHelper.checkTransaction - 28k
- FieldInfo.<init> - 25k
- DataSetManager.register - 19k
- TableMapper$PermanentTableMapper.getDMOClass - 17k

String.toUpperCase usage - 1.47 million calls:

- CompareOps.equals (from application runtime) - 535k
- I18nOps.chr - 250k
- h2.Function.getSimpleValue - 178k
- ScopedSymbolDictionary.processKey - 116k (from lookup, addEntry, locate, used from SharedVariableManager and TRPL dynamic query
- Text.hashCode - 90k (from FastFindCache$L2Key and P2JField.computeHash)
- BaseDataType$Type.of - 79k
- BufferType.<init> - 56k
- Validation.setUniqueQueryParameters - 36k

#5 - 10/26/2022 08:53 AM - Constantin Asofiei

In a standalone test like this:

```
   private static void testStringLowerUpperCode(String s, int n, boolean upper)
   {
      long n1 = System.nanoTime();
      for (int i = 0; i < n; i++)
      {
         if (upper)
         {
            s.toUpperCase();
         }
         else
         {
            s.toLowerCase();
         }
      }
      long n2 = System.nanoTime();
      System.out.println((n2 - n1) / 1000000d);
   }
```

these calls:

```
      testStringLowerUpperCode("ABCDEFGHIJKLMN", 2100000, false);
      testStringLowerUpperCode("abcdefghijklmn", 1400000, true);
```

show:

```
127.055793
78.105617
```

Reducing this will save both CPU power on the actual lower/upper transformation, and heap because another String instance will not be created.

#6 - 01/04/2023 11:22 AM - Greg Shah

- Assignee set to Hynek Cihlar

#7 - 01/16/2023 03:52 AM - Hynek Cihlar

- Status changed from New to WIP

#8 - 01/18/2023 08:56 AM - Hynek Cihlar

- % Done changed from 0 to 90

6129b revision 14367 resolves most of the cases reported in #6820-4.

Some cases were kept because the use of lower/upper string casing was valid, some were kept because the removal of the lower/upper casing was far too complex. More details follows.

Please review.

#9 - 01/18/2023 10:16 AM - Constantin Asofiei

Hynek, you add a dependency on commons-collections4 ver 4.4, but FWD already is (in)directly dependent on commons-collection-3.2.2 - so there is a conflict here. Also, can you work without AbstractHashedMap in CaseInsensitiveHashMap and CaseInsensitiveLinkedHashMap?

Also, you've changed cases like:

```
            name = TextOps.rightTrimLower(name);

            RecordBuffer rb = bm.lookupByName(name);
```

to

```
            name = TextOps.rightTrimNative(name);

            RecordBuffer rb = bm.lookupByName(name);
```

This is OK only if bm.lookupByName(name); does case-insensitive lookup. Please double-check these rightTrimLower changed to rightTrimNative cases.

#10 - 01/18/2023 10:41 AM - Hynek Cihlar

Constantin Asofiei wrote:

> Hynek, you add a dependency on commons-collections4 ver 4.4, but FWD already is (in)directly dependent on commons-collection-3.2.2 - so there is a conflict here.

Actually we explicitly declare dependency on commons-collection-3.2.2. I deliberately added commons-collections4 as it reimplements the collections with generics and so can directly replace java collections. On the other hand I didn't want to change all the places where version 3 is used, and instead do it more incrementally. Both 3 and 4 can live side by-side, unless I'm mistaken. Or do you see any issues there?

> Also, can you work without AbstractHashedMap in CaseInsensitiveHashMap and CaseInsensitiveLinkedHashMap?

I'm not sure we want to give up AbstractHashedMap. My preliminary testing shows it outperforms Java hash map and hash set. I will post more details later.

> Also, you've changed cases like:
> [...]
> to
> [...]
>
> This is OK only if bm.lookupByName(name); does case-insensitive lookup. Please double-check these rightTrimLower changed to rightTrimNative cases.

All the places where rightTrimNative replaces rightTrimLower does handle the related logic. The case you posted above is covered.

#11 - 01/18/2023 11:09 AM - Eric Faulhaber

Hynek Cihlar wrote:

> I'm not sure we want to give up AbstractHashedMap. My preliminary testing shows it outperforms Java hash map and hash set. I will post more details later.

This is the critical point I was going to ask about. With the sweeping changes we are making here, we have to be absolutely sure that any new data structures or library dependencies actually are more performant than the existing implementations. I am keen to see those details (not just for AbstractHashedMap, but any other new APIs we are using). Thanks.

#12 - 01/18/2023 11:12 AM - Constantin Asofiei

Hynek Cihlar wrote:

> Actually we explicitly declare dependency on commons-collection-3.2.2. I deliberately added commons-collections4 as it reimplements the collections with generics and so can directly replace java collections. On the other hand I didn't want to change all the places where version 3 is used, and instead do it more incrementally. Both 3 and 4 can live side by-side, unless I'm mistaken. Or do you see any issues there?

Both commons-collections-3.2.2.jar and commons-collections4-4.4.jar are copied in build/lib. And which version will get loaded is just a matter of classpath.

I don't like using two versions of the same library - we will get in trouble at some point.

#13 - 01/18/2023 12:25 PM - Constantin Asofiei

Unfortunately this introduces a visible slowdown in a large customer application performance testing - ~200ms or so.

#14 - 01/18/2023 02:06 PM - Hynek Cihlar

Constantin Asofiei wrote:

> Unfortunately this introduces a visible slowdown in a large customer application performance testing - ~200ms or so.

Strange, I've seen consistently better results in the range of several hundred of milliseconds. What Java do you use? Please run the main methods in CaseInsensitiveHashMap, CaseInsensitiveLinkedHashMap and CaseInsensitiveHashSet and report the results.

#15 - 01/18/2023 02:21 PM - Constantin Asofiei

Hynek Cihlar wrote:

> Constantin Asofiei wrote:
>
> > Unfortunately this introduces a visible slowdown in a large customer application performance testing - ~200ms or so.
>
> Strange, I've seen consistently better results in the range of several hundred of milliseconds. What Java do you use? Please run the main methods in CaseInsensitiveHashMap, CaseInsensitiveLinkedHashMap and CaseInsensitiveHashSet and report the results.

The results are as expected, collections4 is a lot faster than HashMap.

But I noticed something: there is this code in BufferImpl.java line 11012, this uses the wrong key:

```
                              DataSource.FieldReference srcField =
                                    after.dataSrcMapping.get(new CaseInsensitiveString(fldName));
```

dataSrcMapping is a CaseInsensitiveLinkedHashMap, with String type for key. Something else to note: these maps need to force the key to be a String, and not anything. Please fix this.

#16 - 01/18/2023 02:36 PM - Hynek Cihlar

Constantin Asofiei wrote:

> Both commons-collections-3.2.2.jar and commons-collections4-4.4.jar are copied in build/lib. And which version will get loaded is just a matter of classpath.
>
> I don't like using two versions of the same library - we will get in trouble at some point.

Please note all classes of version 3 are located in the package org.apache.commons.collections while version 4 in the package org.apache.commons.collections4. Thus both jars may be loaded side by side.

#17 - 01/18/2023 02:45 PM - Hynek Cihlar

Constantin Asofiei wrote:

> Hynek Cihlar wrote:
>
> > Constantin Asofiei wrote:
> >
> > > Unfortunately this introduces a visible slowdown in a large customer application performance testing - ~200ms or so.
> >
> > Strange, I've seen consistently better results in the range of several hundred of milliseconds. What Java do you use? Please run the main methods in CaseInsensitiveHashMap, CaseInsensitiveLinkedHashMap and CaseInsensitiveHashSet and report the results.
>
> The results are as expected, collections4 is a lot faster than HashMap.
>
> But I noticed something: there is this code in BufferImpl.java line 11012, this uses the wrong key:
> [...]

Good find! I fixed this in 6129b revision 14368. There was 4 occurrences of the wrong use of CaseInsensitiveString for the key.

> Something else to note: these maps need to force the key to be a String, and not anything. Please fix this.

The maps do allow any object as the key as the overriden hash and isEqualKey handle any object type for the key, but they use the string representation of the objects as the input for the hash calculation and key equality. So technically any object can be used as the key but string makes the most case obviously.

#18 - 01/18/2023 03:23 PM - Constantin Asofiei

XmlImport.tablesByXmlName can be switched to a CaseInsensitiveHashMap. The other cas remaining - from TempTableHelper constructor, Map<CaseInsensitiveString, String> explicitIndexes = new TreeMap<>();, I don't think it can be switched, as this looks like it requires an ordered map.

#19 - 01/18/2023 03:49 PM - Hynek Cihlar

In the large customer application I see an improvement between 100-200ms between 6129b revisions 14366 and 14368.

#20 - 01/19/2023 06:33 AM - Hynek Cihlar

XmlImport.tablesByXmlName changed to CaseInsensitiveHashMap and some other cases in 6129b revision 14369. And related fixes. Please review.

#21 - 01/19/2023 06:48 AM - Hynek Cihlar

Eric Faulhaber wrote:

> Hynek Cihlar wrote:
>
> > I'm not sure we want to give up AbstractHashedMap. My preliminary testing shows it outperforms Java hash map and hash set. I will post more details later.
>
> This is the critical point I was going to ask about. With the sweeping changes we are making here, we have to be absolutely sure that any new data structures or library dependencies actually are more performant than the existing implementations. I am keen to see those details (not just for AbstractHashedMap, but any other new APIs we are using). Thanks.

I introduced three new collections, CaseInsensitiveHashMap, CaseInsensitiveLinkedHashMap and CaseInsensitiveHashSet. The collections compare their keys case-insensitively, i.e. ciMap.get("Hello") == ciMap.get("hello"). They are based on Apache Commons Collections version 4, classes org.apache.commons.collections4.map.AbstractHashedMap and org.apache.commons.collections4.map.LinkedMap (also a hash map). The purpose for these is to improve performance over the approach widely used in FWD, i.e. lower/upper casing the key with the native Java hash maps.

Speaking of performance. According to the micro benchmarks in the main methods of the introduced collections, and on my system, the put operation is about 120% faster, while get is about 25% faster.

There don't seem to be any other downsides for using the new collections. They expose the same public interfaces as the native Java collections, and they use the same defaults. Thus they seem to be good candidates for any current FWD use case.

When comparing performance of the plain Apache Commons maps to the native Java, the Apache commons are again significantly faster for put (about 10-200% faster depending on the size of the map, the higher the number of entries the bigger the improvement) but slightly slower for get (about 4-7% slower).

#22 - 01/19/2023 11:21 AM - Hynek Cihlar

- Status changed from WIP to Review

- % Done changed from 90 to 100

I removed all the upper/lower case occurrences in the list in #6820-4 with the exception of the following list. These items didn't make it because the use of upper/lower is legal or the change would be far too complex or the occurrence was in the H2 jar.

```
FieldInfo.<init> - 25k
P2JField.computeHash - 90k
h2.Function.getSimpleValue - 178k
ScopedSymbolDictionary.processKey - 116k (from lookup, addEntry, locate, used from SharedVariableManager and T
RPL dynamic query)
Validation.setUniqueQueryParameters - 36k
```

For CompareOps.equals (from application runtime) - 535k I didn't find any reference to toUpperCase or toLowerCase.

#23 - 01/19/2023 02:20 PM - Constantin Asofiei

Hynek Cihlar wrote:

> ... 6129b revision 14369. And related fixes. Please review.

I see only one issue:

- TempTableBuilder - indexes map was not changed to a case-insensitive map, and there are changes of rightTrimLower to rightTrimNative for this map's key.

#24 - 01/19/2023 02:33 PM - Hynek Cihlar

Constantin Asofiei wrote:

> Hynek Cihlar wrote:
>
> > ... 6129b revision 14369. And related fixes. Please review.
>
> I see only one issue:
>
> - TempTableBuilder - indexes map was not changed to a case-insensitive map, and there are changes of rightTrimLower to rightTrimNative for this map's key.

Resolved in 6129b revision 14374.

#25 - 01/20/2023 09:48 AM - Hynek Cihlar

I marked the issue at 100%. But there are still some left out places, those that were more complex to address (see #6820-22). If we get to an agreement these (or some of these) should be addressed, too, the % Done should be decreased accordingly.

#26 - 01/23/2023 02:46 PM - Hynek Cihlar

- % Done changed from 100 to 80

#27 - 01/26/2023 09:58 AM - Hynek Cihlar

The use of toLowerCase and toUpperCase in h2.Function.getSimpleValue - 178k is valid there.

#28 - 01/26/2023 10:06 AM - Hynek Cihlar

- % Done changed from 80 to 100

The use of String.toLowerCase and toUpperCase in P2JField.computeHash - 90k and Validation.setUniqueQueryParameters - 36k is valid.

The removal of toLowerCase and toUpperCase in FieldInfo.<init> - 25k and ScopedSymbolDictionary.processKey - 116k will require a huge amount of effort compared to the performance gains. I don't think it makes sense to proceed with the changes here.

#29 - 02/13/2023 08:52 AM - Constantin Asofiei

Hynek, what times do you get for StringHelper.hashCodeCaseInsensitive test? Please test what happens if you use something like this:

```
   private static final int[] UPPER_CASE = new int[65535];

   static
   {
      Arrays.fill(UPPER_CASE, -1);
   }

   /**
    * Calculates a case-insensitive hash of the supplied string.
    *
    * @param   s
    *          A string value. Must not be {@code null}.
    *
    * @return  a hash value.
    */
   public static int hashCodeCaseInsensitive(String s)
   {
      int hash = 0;
      int sz = s.length();
      for (int i = 0; i < sz; i++)
      {
         char ch = s.charAt(i);

         int ich = (int) ch;
         int uch = UPPER_CASE[ich];

         if (uch == -1)
         {
            synchronized (UPPER_CASE)
            {
               ch = Character.toUpperCase(ch);
               UPPER_CASE[ich] = ch;
            }
         }
         else
         {
            ch = (char) (uch & 0xFFFF);
         }

         // ch = Character.toUpperCase(ch);
         hash = hash * 31 + ch;
      }
      return hash;
   }
```

where we cache the upper-case counterpart, to avoid calling Character.toUpperCase().

#30 - 02/13/2023 09:09 AM - Hynek Cihlar

Constantin Asofiei wrote:

> Hynek, what times do you get for StringHelper.hashCodeCaseInsensitive test? Please test what happens if you use something like this:

Good idea!

Here are the numbers. Tested with StringHelper.main. The results are in ns.

```
Java native: 5189776196
FWD orig: 5785375185
FWD without toUpperCase: 6243106437
FWD without toUpperCase pre-filled: 5360210731 
```

The FWD without toUpperCase pre-filled is as follows:

```
   private static final int[] UPPER_CASE = new int[65535];

   static
   {
      Arrays.fill(UPPER_CASE, -1);
      for (char i = 0; i < UPPER_CASE.length; i++)
      {
         UPPER_CASE[i] = Character.toUpperCase(i);
      }
   }

   public static int hashCodeCaseInsensitive(String s)
   {
      int hash = 0;
      int sz = s.length();
      for (int i = 0; i < sz; i++)
      {
         char ch = s.charAt(i);

         int uch = UPPER_CASE[ch];
         ch = (char) (uch & 0xFFFF);

         // ch = Character.toUpperCase(ch);
         hash = hash * 31 + ch;
      }
      return hash;
   }
```

#31 - 02/13/2023 09:12 AM - Hynek Cihlar

And some cleanup:

```
   private static final char[] UPPER_CASE = new char[65535];

   static
   {
      for (char i = 0; i < UPPER_CASE.length; i++)
      {
         UPPER_CASE[i] = Character.toUpperCase(i);
      }
   }

   public static int hashCodeCaseInsensitive(String s)
   {
      int hash = 0;
      int sz = s.length();
      for (int i = 0; i < sz; i++)
      {
         char ch = s.charAt(i);
         hash = hash * 31 + UPPER_CASE[ch];
      }
      return hash;
   }
```

#32 - 02/13/2023 09:28 AM - Constantin Asofiei

Hynek Cihlar wrote:

> And some cleanup:

Great, I'll test this and include it in my larger changes.

#33 - 02/15/2023 04:17 AM - Constantin Asofiei

Committed to 7026a rev 14810.