What is GS Collections?

• Open source Java collections framework developed in Goldman Sachs
  – In development since 2004
  – Hosted on GitHub w/ Apache 2.0 License
    • [github.com/goldmansachs/gs-collections](https://github.com/goldmansachs/gs-collections)

• GS Collections Kata
  – Internal training developed in 2007
  – Taught to > 1,500 GS Java developers
  – Hosted on GitHub w/ Apache 2.0 License
    • [github.com/goldmansachs/gs-collections-kata](https://github.com/goldmansachs/gs-collections-kata)
1. Extensive use of collections across all of our Java code
2. Fast response times needed so collections in memory
   – Optimizations for both space and speed
3. Constantly changing environment
   – Less boilerplate code to maintain is more desirable so we can focus on business opportunities
4. Usage of patterns improves code review speed and quality
   – Great support in modern IDE’s for pattern based operations
5. Needed features from all of the major collection libraries
   – We get to have that without having to manage the vastness of that diversification
Why do we use GS Collections?

1. Object-Oriented Design + Functional API
2. Rich Lambda-Ready Iteration API
   – Fluent API that works great with Java 8
3. Memory Efficient Containers
4. Completely compatible with JDK interfaces
5. Additional Object Containers not in JDK
6. Primitive Containers
7. Immutable Containers
8. Parallel Eager and Lazy iteration
9. Similar to Smalltalk (api) and Scala (hierarchy)
Questions for this group

• Do you use GS Collections today?
• Is there anything that keeps you from using GS Collections in your code bases?
• What are our plans for Java 9/10 for the Java Collections framework?
  – Will anyone sponsor a JSR for Collections 2?
  – Should GS sponsor a JSR for Collections 2?
• Should a Java Collections 2 implementation “Rock Hard or Go Home”?
  – Full primitive support, immutable containers, multimaps, bags, bimaps, eager/lazy, serial/parallel, etc.
# Framework Comparisons

<table>
<thead>
<tr>
<th>Features</th>
<th>GSC 5.0</th>
<th>Java 8</th>
<th>Guava</th>
<th>Trove</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rich API</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td>☑</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Readable, Mutable, Immutable, FixedSize, Lazy</td>
<td>mutable, Stream</td>
<td>Mutable, Fluent</td>
<td>Mutable</td>
<td>Readable, Mutable, Immutable, Lazy</td>
</tr>
<tr>
<td>Optimized Set &amp; Map</td>
<td>☑ (+Bag)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immutable Collections</td>
<td>☑</td>
<td></td>
<td>☑</td>
<td></td>
<td>☑</td>
</tr>
<tr>
<td>Primitive Collections</td>
<td>☑ (+Bag, +Immutable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multimaps</td>
<td>☑ (+Bag, +SortedBag)</td>
<td>☑ (+Linked)</td>
<td></td>
<td>(Multimap trait)</td>
<td></td>
</tr>
<tr>
<td>Bags (Multisets)</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BiMaps</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration Styles</td>
<td>Eager/Lazy, Serial/Parallel</td>
<td>Lazy, Serial/Parallel</td>
<td>Lazy, Serial</td>
<td>Eager, Serial</td>
<td>Eager/Lazy, Serial/Parallel (Lazy Only)</td>
</tr>
</tbody>
</table>
GS Collections: Design Concepts

RichIterable

Readable Interface

Mutable Interface

Immutable Interface

Bag, Set, List, Stack, Map, etc.

MutableBag, MutableList, etc.

ImmutableBag, ImmutableList, etc.
### RichIterable interface: 50+ Unique Methods, 75+ Total Methods (w/overloads)

<table>
<thead>
<tr>
<th>Method</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregateBy</td>
<td>λ aggregateBy</td>
<td>notEmpty</td>
</tr>
<tr>
<td>aggregateInPlaceBy</td>
<td>λ aggregateInPlaceBy</td>
<td>λ partition</td>
</tr>
<tr>
<td>allSatisfy</td>
<td>λ allSatisfy</td>
<td>λ reject</td>
</tr>
<tr>
<td>anySatisfy</td>
<td>λ anySatisfy</td>
<td>λ rejectWith</td>
</tr>
<tr>
<td>appendString</td>
<td>λ appendString</td>
<td>λ select</td>
</tr>
<tr>
<td>asLazy</td>
<td>λ asLazy</td>
<td>λ selectWith</td>
</tr>
<tr>
<td>chunk</td>
<td>λ chunk</td>
<td>λ size</td>
</tr>
<tr>
<td>collect</td>
<td>λ collect</td>
<td>λ toArray</td>
</tr>
<tr>
<td>collectIf</td>
<td>λ collectIf</td>
<td>λ toBag</td>
</tr>
<tr>
<td>collectWith</td>
<td>λ collectWith</td>
<td>λ toList</td>
</tr>
<tr>
<td>contains</td>
<td>λ contains</td>
<td>λ toMap</td>
</tr>
<tr>
<td>containsAll</td>
<td>λ containsAll</td>
<td>λ toSet</td>
</tr>
<tr>
<td>containsAllArguments</td>
<td>λ containsAllArguments</td>
<td>λ toSortedList / toSortedListBy</td>
</tr>
<tr>
<td>containsAllIterable</td>
<td>λ containsAllIterable</td>
<td>λ toSortedMap</td>
</tr>
<tr>
<td>count</td>
<td>λ count</td>
<td>λ toSortedSet / toSortedSetBy</td>
</tr>
<tr>
<td>detect</td>
<td>λ detect</td>
<td>zip</td>
</tr>
<tr>
<td>detectIfNone</td>
<td>λ detectIfNone</td>
<td>zipWithIndex</td>
</tr>
</tbody>
</table>
## Eager versus Lazy Iteration

<table>
<thead>
<tr>
<th>Iteration Style</th>
<th>GSC Example</th>
<th>JDK 8 Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Eager (collect)</td>
<td>List&lt;Address&gt; addresses = people.collect(Person::*getAddress);</td>
<td></td>
</tr>
<tr>
<td>Serial Lazy (collect / map)</td>
<td>LazyIterable&lt;Address&gt; addresses = people.asLazy()</td>
<td>Stream&lt;Address&gt; addresses = people.stream()</td>
</tr>
<tr>
<td></td>
<td>.collect(Person::*getAddress);</td>
<td>.map(Person::*getAddress);</td>
</tr>
<tr>
<td>Serial Lazy (collect / map, toList)</td>
<td>List&lt;Address&gt; addresses = people.asLazy()</td>
<td>List&lt;Address&gt; addresses = people.stream()</td>
</tr>
<tr>
<td></td>
<td>.collect(Person::*getAddress) .toList();</td>
<td>.map(Person::*getAddress) .collect(Collectors.toList());</td>
</tr>
<tr>
<td>Parallel Eager</td>
<td>Collection&lt;Address&gt; addresses = ParallelIterate.collect(</td>
<td></td>
</tr>
<tr>
<td></td>
<td>people, Person::*getAddress);</td>
<td></td>
</tr>
<tr>
<td>Parallel Lazy</td>
<td>ParallelListIterable&lt;Address&gt; addresses = people.asParallel(executor, batchSize)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.collect(Person::*getAddress);</td>
<td>Stream&lt;Address&gt; addresses = people.parallelStream()</td>
</tr>
<tr>
<td></td>
<td>.map(Person::*getAddress);</td>
<td>.map(Person::*getAddress);</td>
</tr>
</tbody>
</table>
GS Collections in terms of lines of code

- 295,036 lines of Java code
- 9,959 lines of Scala code
- 76,240 lines of StringTemplate templates

- Total Java code after code generation:
  - \(1,438,477\) lines of Java code your developers don’t have to write and can use for free
Reducing code using lambdas

• Converted nearly all of our anonymous inner classes in tests to lambdas and method references.
• 9% reduction of code in our unit test module
  – Dropped from 105,206 LOC to 95,775
• Many examples of code savings from Java 8 can be found in our unit test suite module in GitHub.
  – Possibly one of the first large scale usages of Java 8 lambdas and method references in a code base
• UnifiedMap – built without using Entry objects.
• UnifiedSet – not built using a map.
• Empty should be empty.
• Primitive Collections.
• Memory efficient containers for small-sized collections (>1 and < 11).
For every put, HashMap creates an Entry object.

UnifiedMap stores keys and values in alternate slots on a single array.

Consecutive memory locations are faster to access.
Save 50% Memory with the GSC UnifiedMap

Mutable Map

Size (Mb) vs Elements

- JDK HashMap
- GSC UnifiedMap
- Trove THashMap
- JDK Hashtable

© 2014 Goldman Sachs. All rights reserved.
• HashSet uses HashMap as its backing collection.
• For every add, an Entry object is created with the element as key and null value.
• UnifiedSet uses an array as its backing collection.
Save 400% Memory with the GSC UnifiedSet

![Mutable Set Diagram]

- JDK HashSet
- GSC UnifiedSet
- Trove THashSet
Primitive Collections

• What are primitives?
  • Not everything in java is an Object.
  • Primitive types are *automatic* variables that are not references.
  • The variables hold the value and its place on the stack, so it’s much more efficient.

• Why primitive collections?
  • Reduced memory usage
  • Improved performance
  • Eliminates the need to depend on multiple libraries – PCJ, Trove etc.

• What primitive collections are available in GSC?
  • List, Set, Map, Stack, Bag

• For what all primitive types?
  • All eight: boolean, byte, char, double, float, int, long, short
Save Memory with Primitive Collections: IntList

IntList

Size (Mb)

0 5 10 15 20 25

Elements

JDK ArrayList
GSC IntArrayList
Trove TIntArrayList

4/26/2014 © 2014 Goldman Sachs. All rights reserved.
Questions for this group

• Do you use GS Collections today?
• Is there anything that keeps you from using GS Collections in your code bases?
• What are our plans for Java 9/10 for the Java Collections framework?
  – Will anyone sponsor a JSR for Collections 2?
  – Should GS sponsor a JSR for Collections 2?
• Should a Java Collections 2 implementation “Rock Hard or Go Home”?
  – Full primitive support, immutable containers, multimaps, bags, bimaps, eager/lazy, serial/parallel, etc.
Appendix

- GS Collections Class Hierarchy
- GS Collections Java 8 Code examples
GS Collections adds 30 interfaces to enhance the 5 basic JDK Collections interfaces
GS Collections adds 12 interfaces to enhance the 3 basic JDK Map interfaces

Note: GS Collections Maps are RichIterable on their values
GS Collections adds 22 interfaces to support different forms of Multimaps
GS Collections: PrimitiveIterable Hierarchy

GS Collections adds 16 interfaces x 8 primitive types
( boolean, byte, char, double, float, int, long, short )
GS Collections: ParallelIterable Hierarchy
@Test
public void getCustomerNames()
{
    Function<Customer, String> fn = new Function<Customer, String>()
    {
        @Override
        public String valueOf(Customer customer)
        {
            return customer.getName();
        }
    };

    /**
     * Get the name of each of the company's customers.
     */
    MutableList<Customer> customers = this.company.getCustomers();
    MutableList<String> names = customers.collect(fn);
    MutableList<String> expectedNames =
        FastList.newListWith("Fred", "Mary", "Bill");
    Assert.assertEquals(expectedNames, names);
}
@Test
public void getCustomerNames()
{
    /**
     * Get the name of each of the company's customers.
     */
    MutableList<Customer> cs = this.company.getCustomers();
    MutableList<String> names = cs.collect(Customer::getName);

    MutableList<String> expected =
        FastList.newListWith("Fred", "Mary", "Bill");
    Assert.assertEquals(expected, names);
}

* Saved 8 lines of code with method reference *
@Test
def filterOrderValues():
    orders = this.company.getMostRecentCustomer().getOrders()
    DoubleList filtered = this.company.getMostRecentCustomer().
        .getOrders().
        asLazy().
        collectDouble(Order.TO_VALUE). 
        select(DoublePredicates.greaterThan(1.5)). 
        toSortedList();
    Assert.assertEquals(DoubleArrayList.newListWith(1.75, 372.5), filtered);
}

class Order:
    public static final DoubleFunction<Order> TO_VALUE = new DoubleFunction<Order>() {
        public double doubleValueOf(Order order)
        {
            return order.getValue();
        }
    };
    ...

@Test
public void filterOrderValues() {
    DoubleList filtered = this.company.getMostRecentCustomer()
        .getOrders()
        .asLazy()
        .collectDouble(Order::getValue)
        .select(value -> value > 1.5)
        .toSortedList();

    Assert.assertEquals(
        DoubleArrayList.newListWith(1.75, 372.5),
        filtered);
}

* Saved 8 lines of code with method reference *
• Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.