Goldman Sachs

GS Collections and Java 8
Functional, Fluent, Friendly & Fun!

GS.com/Engineering
Spring, 2015

Craig Motlin

we
BUILD

Goldman Sachs

Agenda

- Introductions
- Lost and Found
- Streams
- The Iceberg
  - APIs
  - Fluency
  - Memory Efficiency
  - Method references are awesome
- Framework Comparisons

What is GS Collections?

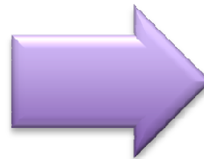
- Open source Java collections framework developed in Goldman Sachs
  - In development since 2004
  - Hosted on GitHub w/ Apache 2.0 License
    - github.com/goldmansachs/gs-collections
- GS Collections Kata
  - Internal training developed in 2007
  - Taught to > 1,500 GS Java developers
  - Hosted on GitHub w/ Apache 2.0 License
    - github.com/goldmansachs/gs-collections-kata

Goldman Sachs

Paradise Lost

1997 - Smalltalk Best Practice Patterns (Kent Beck)

- do:
- select:
- reject:
- collect:
- detect:
- detect:ifNone:
- inject:into:
- …
  (Dr. Seuss API)



2007 - Implementation Patterns (Kent Beck)

- Map
- List
- Set

- The collection iteration patterns disappeared
- All that remained were the types

Goldman Sachs

Paradise Found

Pattern in Smalltalk-80 | Pattern in Classic Java | Pattern in GS Collections w/ Lambdas

detect

```
list detect: [:each | each > 50].
```

```
for (Integer each : list)
  if (each > 50)
    return each;
return null;
```

```
list.detect(each -> each > 50);
```

select

```
list select: [:each | each > 50].
```

```
List<Integer> result = new ArrayList<>();
for (Integer each : list)
  if (each > 50)
    result.add(each);
```

```
list.select(each -> each > 50);
```

reject

```
list reject: [:each | each > 50].
```

```
List<Integer> result = new ArrayList<>();
for (Integer each : list)
  if (each <= 50)
    result.add(v);
```

```
list.reject(each -> each > 50);
```

any satisfy

```
list anySatisfy: [:each | each > 50].
```

```
for (Integer each : list)
  if (each > 50)
    return true;
return false;
```

```
list.anySatisfy(each -> each > 50);
```

all satisfy

```
list allSatisfy: [:each | each > 50].
```

```
for (Integer each : list)
  if (each <= 50)
    return false;
return true;
```

```
list.allSatisfy(each -> each > 50);
```

collect

```
list collect: [:e | e printString].
```

```
List<String> result = new ArrayList<>();
for (Integer each : list)
  result.add(each.toString());
```

```
list.collect(Object::toString);
```

inject into

```
list inject: 3 into: [:x :y | x + y].
```

```
int result = 3;
for (Integer each : list)
  result = result + each;
```

```
list.injectInto(3, Integer::sum);
```

Goldman Sachs

Lazy by any other name

	GS Collections LazyIterable		Java 8 Streams
detect	`Integer result = list.asLazy() .detectIfNone(e -> e > 50, () -> null);`	findAny	`Integer result = list.stream() .filter(e -> e > 50).findFirst().orElse(null);`
select	`LazyIterable<Integer> result = list.asLazy().select(e -> e > 50);`	filter	`Stream<Integer> result = list.stream().filter(e -> e > 50);`
reject	`LazyIterable<Integer> result = list.asLazy().reject(e -> e > 50);`	filter	`Stream<Integer> result = list.stream().filter(e -> e <= 50);`
any satisfy	`boolean any = list.asLazy().anySatisfy(e -> e > 50);`	any Match	`boolean any = list.stream().anyMatch(e -> e > 50);`
all satisfy	`boolean all = list.asLazy().allSatisfy(e -> e > 50);`	all Match	`boolean all = list.stream().allMatch(e -> e > 50);`
collect	`LazyIterable<String> result = list.asLazy().collect(Object::toString);`	map	`Stream<String> result = list.stream().map(Object::toString);`
inject into	`Integer result = list.asLazy().injectInto(3, Integer::sum);`	reduce	`Integer result = list.stream().reduce(3, Integer::sum);`

Goldman Sachs

Eager vs. Lazy

	Eager GS Collections		Java 8 Streams
detect	`Integer result = list.detect(e -> e > 50);`	findAny	`Integer result = list.stream().filter(e -> e > 50).findFirst().orElse(null);`
select	`MutableList<Integer> result = list.select(e -> e > 50);`	filter	`List<Integer> result = list.stream().filter(e -> e > 50).collect(Collectors.toList());`
reject	`MutableList<Integer> result = list.reject(e -> e > 50);`	filter	`List<Integer> result = list.stream().filter(e -> e <= 50).collect(Collectors.toList());`
any satisfy	`boolean any = list.anySatisfy(e -> e > 50);`	any Match	`boolean result = list.stream().anyMatch(e -> e > 50);`
all satisfy	`boolean all = list.allSatisfy(e -> e > 50);`	all Match	`boolean result = list.stream().allMatch(e -> e > 50);`
collect	`MutableList<String> result = list.collect(Object::toString);`	map	`List<String> result = list.stream().map(Object::toString).collect(Collectors.toList());`
inject into	`Integer result = list.injectInto(3, Integer::sum);`	reduce	`Integer result = list.stream().reduce(3, Integer::sum);`

Goldman Sachs

Java 8 Streams

- Great framework that provides feature rich functional API
- Lazy by default
- Supports serial and parallel iteration patterns
- Support for three types of primitive streams
- Extendable through Collector implementations

- Java 8 Streams is the tip of an enormous iceberg

Iceberg dead ahead!

- Eager iteration patterns on Collections
- Covariant return types on collection protocols
- New Collection Types
  - Bag, SortedBag, BiMap, Multimap
- Memory Efficient Set and Map
- Primitive containers
- Immutable containers

Goldman Sachs

Ice is Twice as Nice

	Java 8	GS Collections
Stream vs. LazyIterable Interfaces	5	9
Functional Interfaces	46	298
Object Container Interfaces	11	75
Primitive Container Interfaces	0	309
Stream vs. RichIterable API	47	109
Primitive Stream vs. Iterable API	48 x 3 = 144	38 x 8 = 304
LOC (Streams vs. GSC w/o code gen)	~15k	~400k

Goldman Sachs

More Iteration Patterns

- flatCollect
- partition
- makeString / appendString
- groupBy
- aggregateBy
- sumOf
- sumBy

Goldman Sachs

Futility of Utility

- Utility
  - Easy to extend with new behaviors without breaking existing clients
- API
  - Easy to discover new features
  - Easy to optimize
  - Easy to read from left to right
  - Return types are specific and easy to understand
  - Verb vs. gerund

Goldman Sachs

Joining vs. MakeString

```java
String joined = things.stream()
   .map(Object::toString)
   .collect(Collectors.joining(", "));
```

```java
String joined =
   things.makeString(", ");
```

Goldman Sachs

SummingInt vs. SumOfInt

```
int total = employees.stream().collect(
  Collectors.summingInt(Employee::getSalary));
```

```
long total =
  employees.sumOfInt(Employee::getSalary);
```

GroupingBy vs. GroupBy

```
Map<Department, List<Employee>> byDept =
  employees.stream()
  .collect(Collectors.groupingBy(
    Employee::getDepartment));
```

```
Multimap<Department, Employee> byDept =
  employees.groupBy(Employee::getDepartment);
```

Goldman Sachs

GroupingBy/SummingBy vs. SumBy

```
Map<Department, Integer> totalByDept =
  employees.stream()
  .collect(Collectors.groupingBy(
    Employee::getDepartment,
    Collectors.summingInt(Employee::getSalary)));
```

```
ObjectLongMap<Department> totalByDept =
  employees.sumByInt(
    Employee::getDepartment,
    Employee::getSalary);
```

Goldman Sachs

PartitioningBy vs. Partition

```java
Map<Boolean, List<Student>> passingFailing
  students.stream()
  .collect(Collectors.partitioningBy(
    s -> s.getGrade() >= PASS_THRESHOLD));
```

```java
PartitionList<Student> passingFailing =
  students.partition(
    s -> s.getGrade() >= PASS_THRESHOLD);
```

Goldman Sachs

How do they stack up?

```
getGrade():1290, FastListTest$Student (com.gs.collections.impl.list.mutable)
lambda$partitionLists$5():1255, FastListTest (com.gs.collections.impl.list.mutable)
test():-1, 1632492873 (com.gs.collections.impl.list.mutable.FastListTest$$Lambda$1
lambda$partitioningBy$146():1139, Collectors (java.util.stream)
accept():-1, 963601816 (java.util.stream.Collectors$$Lambda$6)
accept():169, ReduceOps$3ReducingSink (java.util.stream)
forEachRemaining():116, Iterator (java.util)
forEachRemaining():1801, Spliterators$IteratorSpliterator (java.util)
copyInto():512, AbstractPipeline (java.util.stream)
wrapAndCopyInto():502, AbstractPipeline (java.util.stream)
evaluateSequential():708, ReduceOps$ReduceOp (java.util.stream)
evaluate():234, AbstractPipeline (java.util.stream)
collect():499, ReferencePipeline (java.util.stream)
partitionLists():1254, FastListTest (com.gs.collections.impl.list.mutable)
```

```
getGrade():1290, FastListTest$Student (com.gs.collections.impl.list.mutable)
lambda$partitionLists$dcd4a7c$1():1259, FastListTest (com.gs.collections.impl.list.
accept():-1, 238157928 (com.gs.collections.impl.list.mutable.FastListTest$$Lambda
partition():946, RandomAccessListIterate (com.gs.collections.impl.utility.internal)
partition():1011, ListIterate (com.gs.collections.impl.utility)
partition():262, AbstractMutableList (com.gs.collections.impl.list.mutable)
partitionLists():1258, FastListTest (com.gs.collections.impl.list.mutable)
```

Goldman Sachs

Agenda

- Introductions
- Lost and Found
- Streams
- The Iceberg
  - APIs
  - **Fluency**
  - Memory Efficiency
  - Method references are awesome
- Framework Comparisons

Goldman Sachs

Anagram tutorial

http://docs.oracle.com/javase/tutorial/collections/algorithms/

- Start with all words in the dictionary
- Group them by their alphagrams
  - Alphagram contains sorted characters
  - alerts → aelrst
  - stelar → aelrst
- Filter groups containing at least eight anagrams
- Sort groups by number of anagrams (descending)
- Print them in this format

```
11: [alerts, alters, artels, estral, laster, ratels,
salter, slater, staler, stelar, talers]
```

Goldman Sachs

Anagram tutorial

```java
this.getWords()
    .stream()
    .collect(Collectors.groupingBy(Alphagram::new))
    .values()
    .stream()
    .filter(each -> each.size() >= SIZE_THRESHOLD)
    .sorted(Comparator.<List<?>>comparingInt(List::size).reversed())
    .map(each -> each.size() + ": " + each)
    .forEach(System.out::println);
```

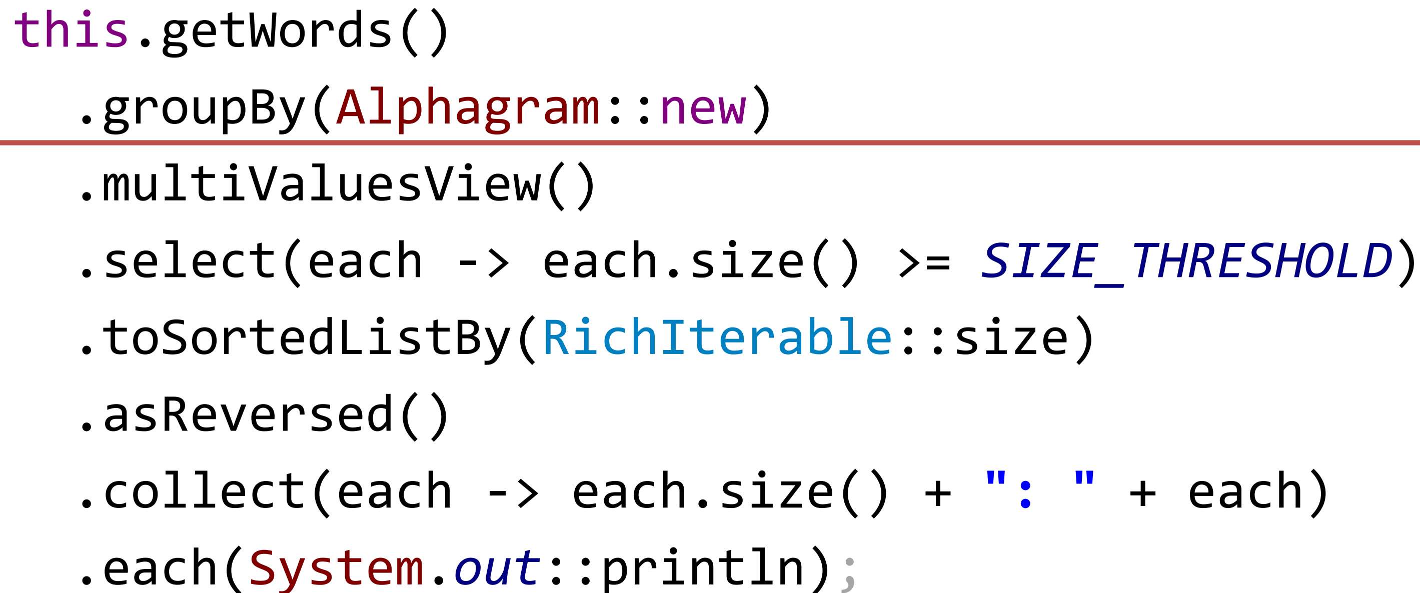
Goldman Sachs

Anagram tutorial

```
this.getWords()
  .groupBy(Alphagram::new)
  .multiValuesView()
  .select(each -> each.size() >= SIZE_THRESHOLD)
  .toSortedListBy(RichIterable::size)
  .asReversed()
  .collect(each -> each.size() + ": " + each)
  .each(System.out::println);
```

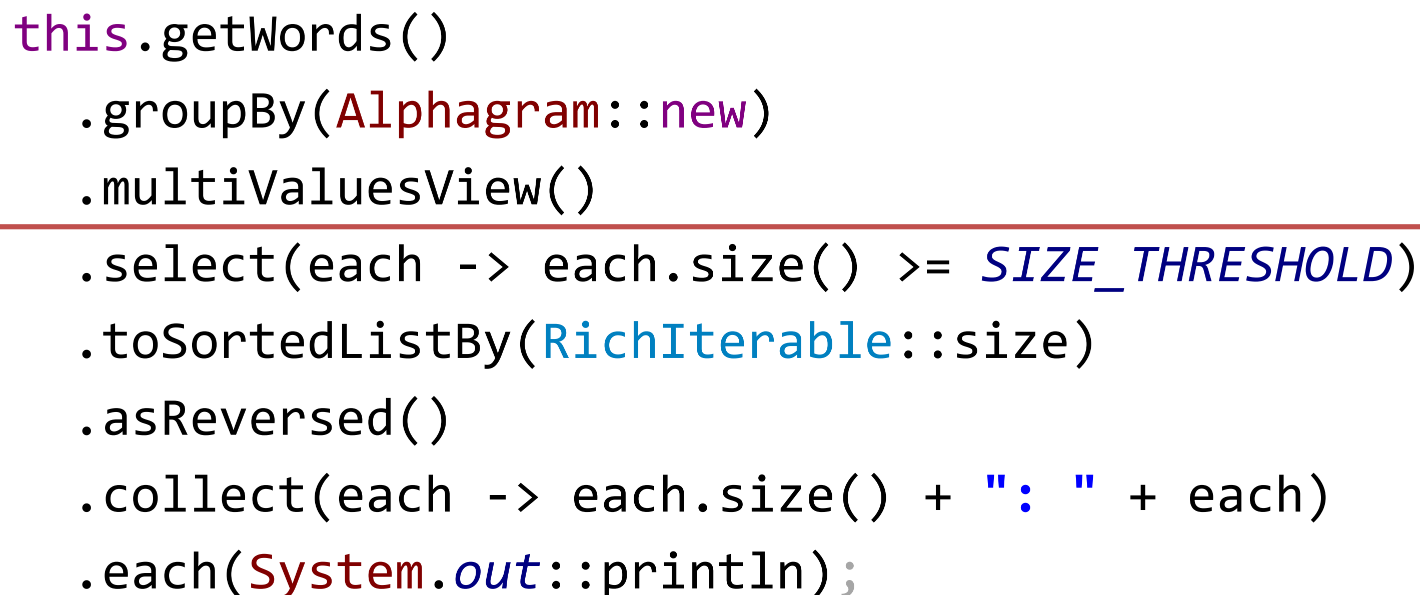
Goldman Sachs

Anagram tutorial



Type: MutableListMultimap<Alphagram, String>

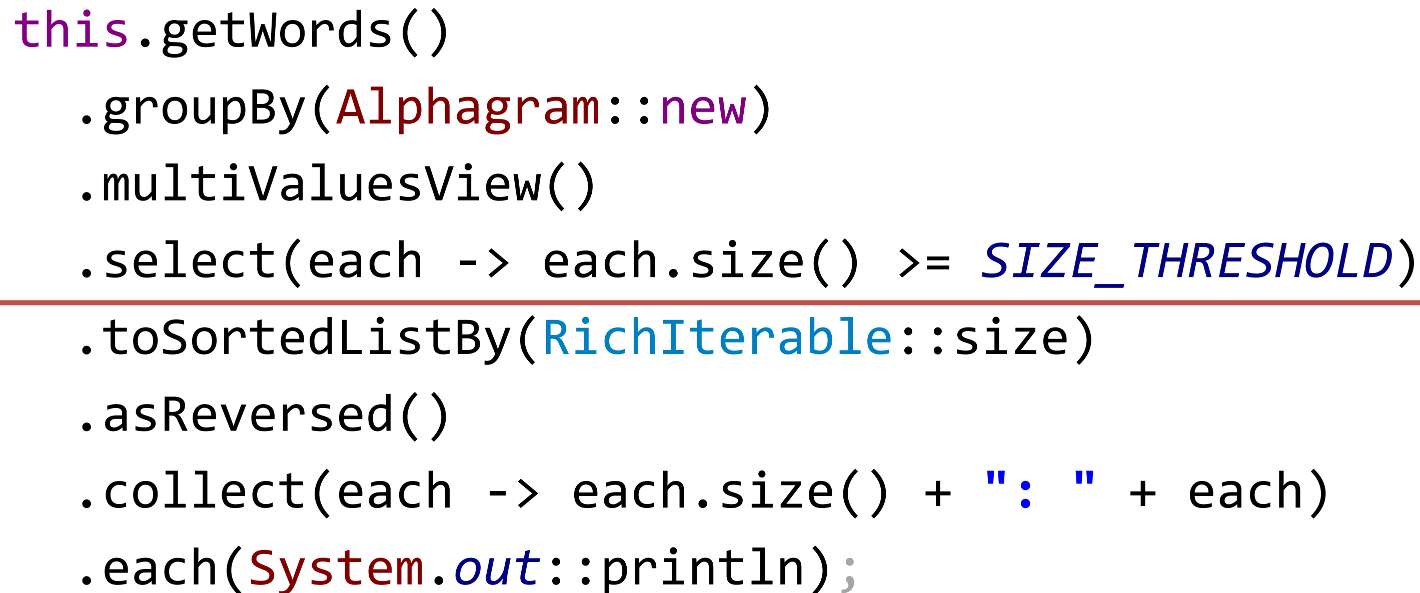
Anagram tutorial



```
this.getWords()
    .groupBy(Alphagram::new)
    .multiValuesView()
    .select(each -> each.size() >= SIZE_THRESHOLD)
    .toSortedListBy(RichIterable::size)
    .asReversed()
    .collect(each -> each.size() + ": " + each)
    .each(System.out::println);
```

Type: `RichIterable<RichIterable<String>>`

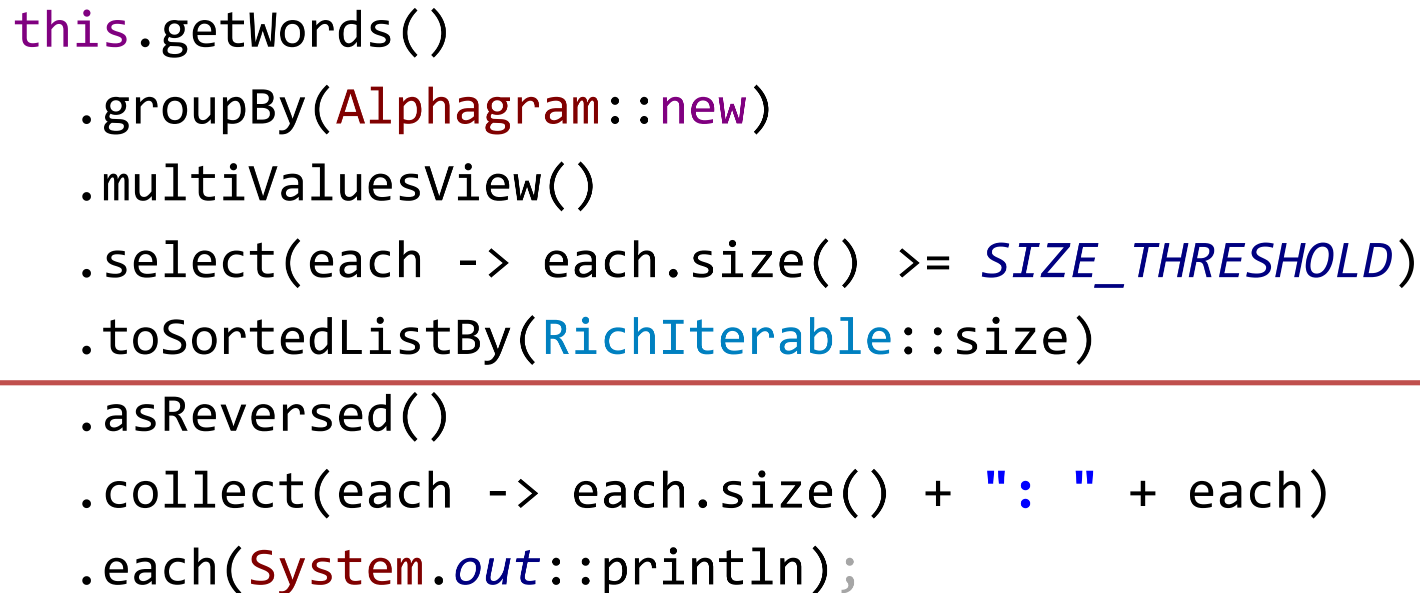
Anagram tutorial



Type: RichIterable<RichIterable<String>>

Goldman Sachs

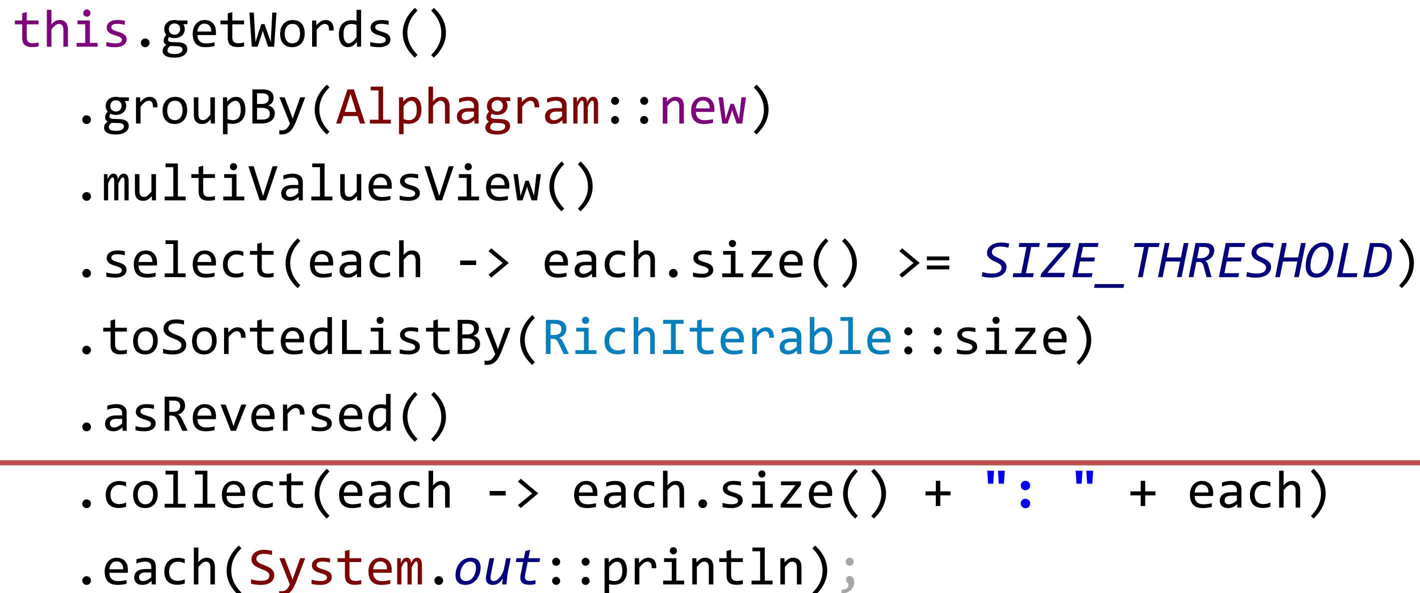
Anagram tutorial



```
this.getWords()
    .groupBy(Alphagram::new)
    .multiValuesView()
    .select(each -> each.size() >= SIZE_THRESHOLD)
    .toSortedListBy(RichIterable::size)
    .asReversed()
    .collect(each -> each.size() + ": " + each)
    .each(System.out::println);
```

Type: `MutableList<RichIterable<String>>`

Anagram tutorial

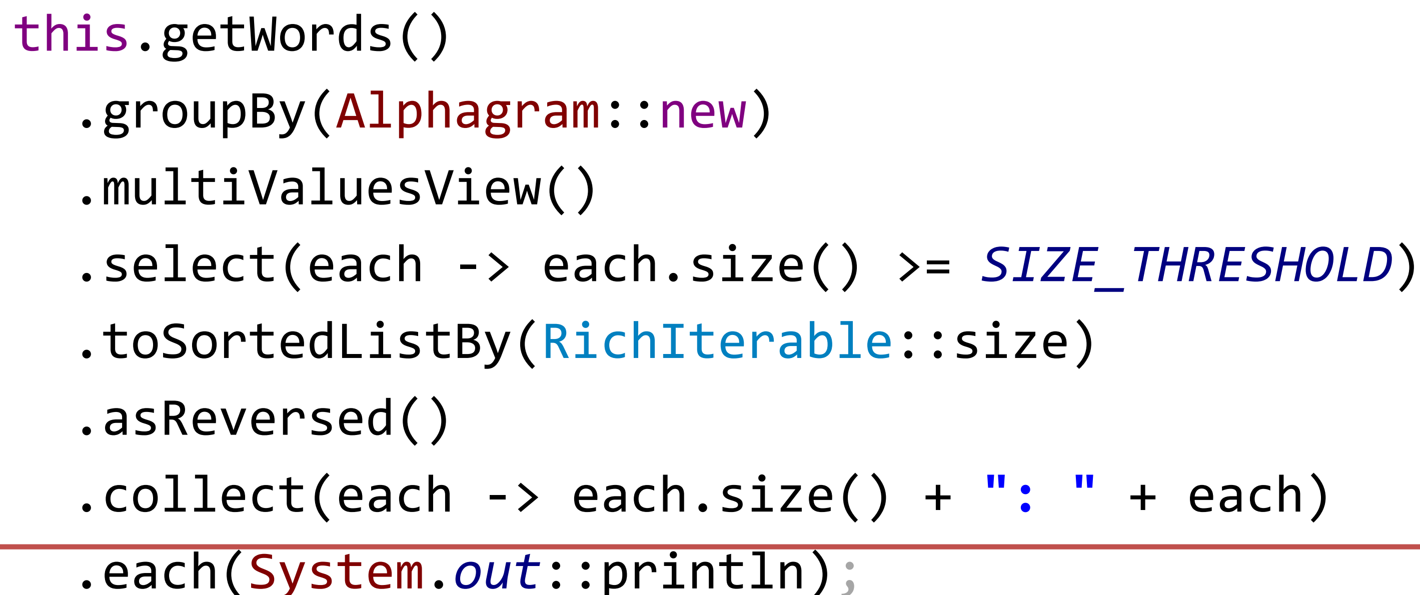


```
this.getWords()
    .groupBy(Alphagram::new)
    .multiValuesView()
    .select(each -> each.size() >= SIZE_THRESHOLD)
    .toSortedListBy(RichIterable::size)
    .asReversed()
    .collect(each -> each.size() + ": " + each)
    .each(System.out::println);
```

Type: `LazyIterable<RichIterable<String>>`

Goldman Sachs

Anagram tutorial



Type: LazyIterable<String>

Parallel Lazy Iteration

```
Stream<Address> addresses =
  people.parallelStream()
    .map(Person::getAddress);
```

```
ParallelListIterable<Address> addresses =
  people.asParallel(executor, batchSize)
    .collect(Person::getAddress);
```

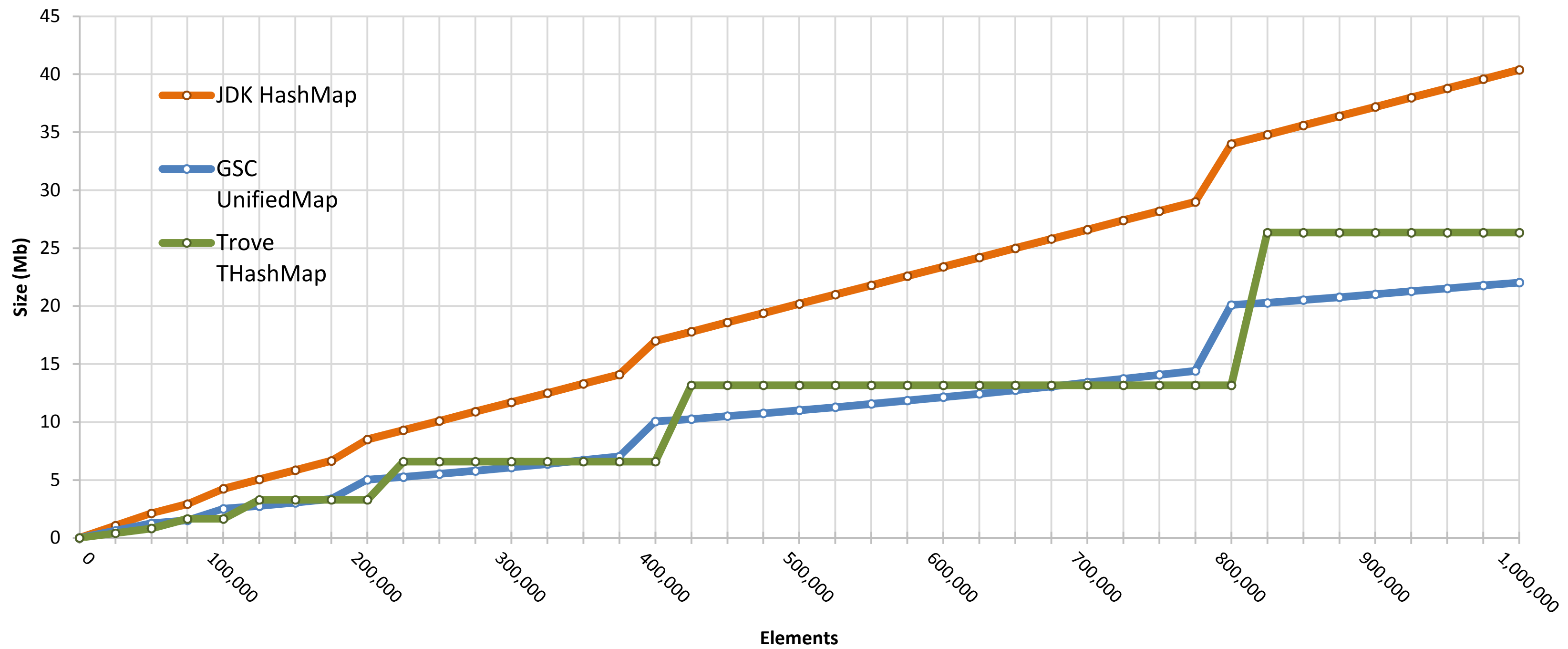
http://www.infoq.com/presentations/java-streams-scala-parallel-collections

Agenda

- Introductions
- Lost and Found
- Streams
- The Iceberg
  - APIs
  - Fluency
  - **Memory Efficiency**
  - Method references are awesome
- Framework Comparisons

Goldman Sachs

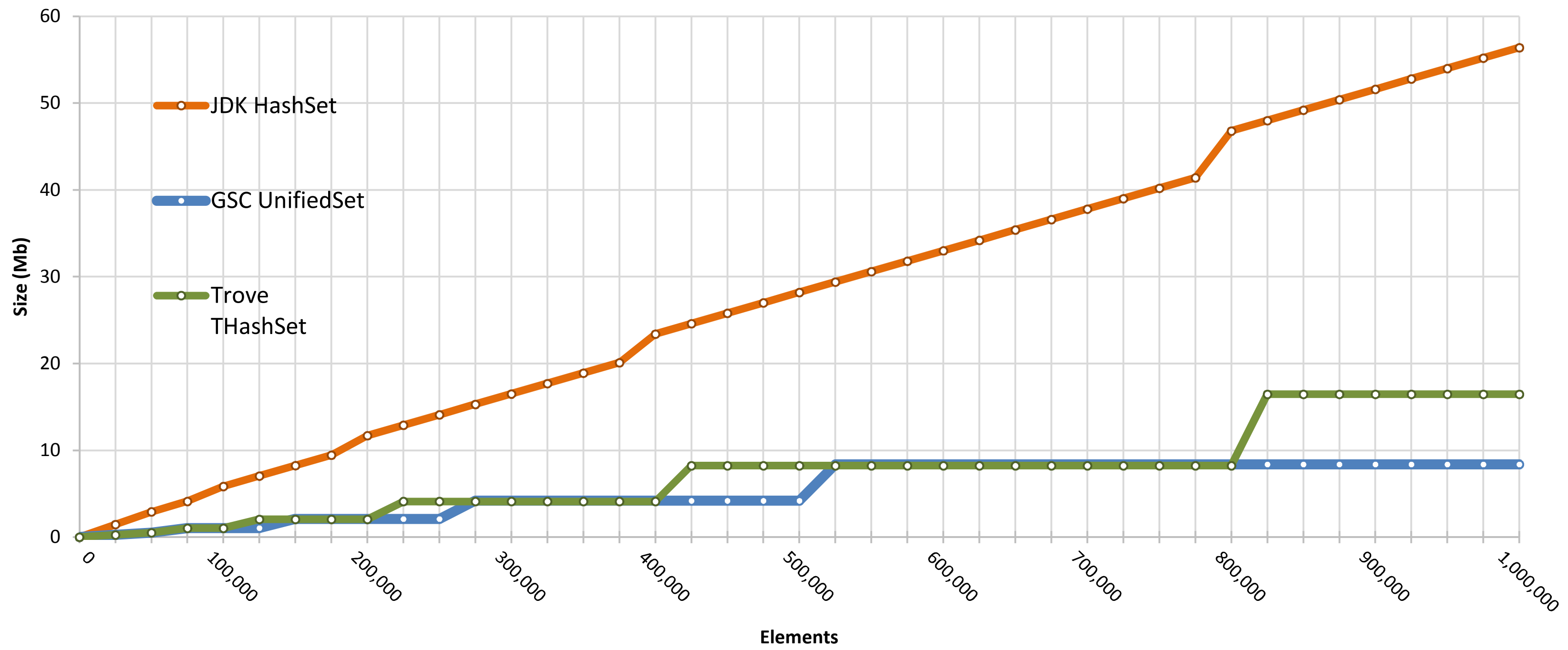
Comparing Maps



- `Entry` holds key, value, next, and hash.
- Better to put the keys and values in the backing array.
- Uses half the memory on average.
- But watch out for `Map.entrySet()`.
  - Leaky abstraction
    - The assumption is that Maps are implemented as tables of Entry objects.
  - It’s now O(n) instead of O(1).
  - Use `forEachKeyValue()` instead.

Goldman Sachs

Comparing Sets



Goldman Sachs

Memory Optimizations

- HashSet is implemented by delegating to a HashMap.
- Entries are still a waste of space.
- Values in each (key, value) pair are a waste of space.
- Uses 4x the memory on average.

Bad decisions from long ago

Map with collision.

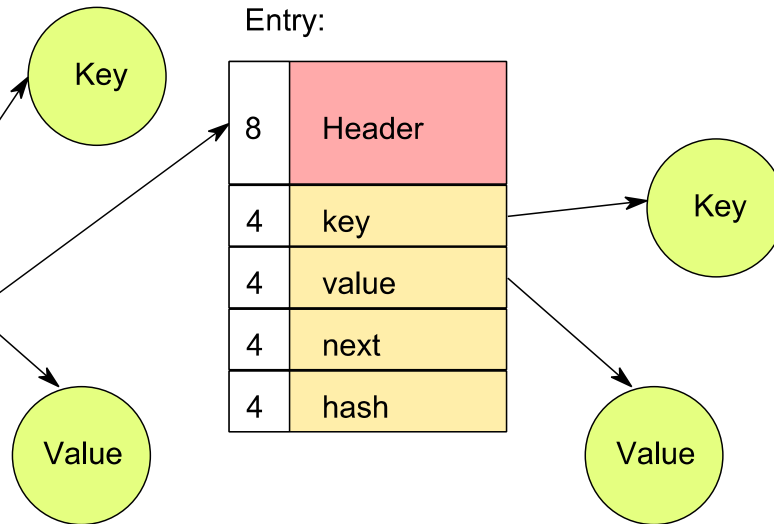
HashMap:

Array of Entry:

8	Header
4	Length
4	Padding
4	(null)
4	(null)
4	reference
4	(null)

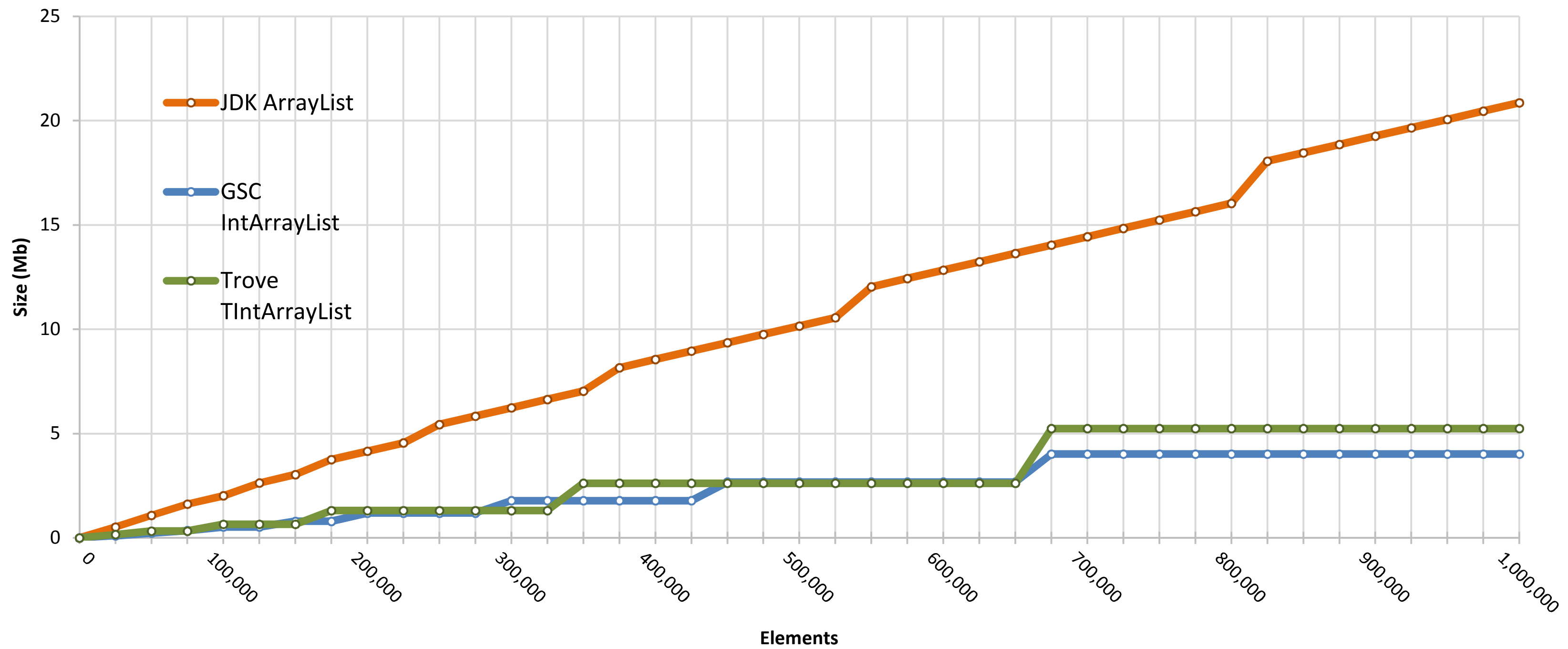
Entry:

8	Header
4	key
4	value
4	next
4	hash



Goldman Sachs

Save memory with Primitive Collections



List<Integer> vs. IntList

- Java has object and primitive arrays
  - Primitive arrays have no behaviors
- Java does not have primitive Lists, Sets or Maps
  - Primitives must be boxed
  - Boxing is expensive
    - Reference + Header + alignment

Agenda

- Introductions
- Lost and Found
- Streams
- The Iceberg
  - APIs
  - Fluency
  - Memory Efficiency
  - **Method references are awesome**
- Framework Comparisons

Lambdas and Method References

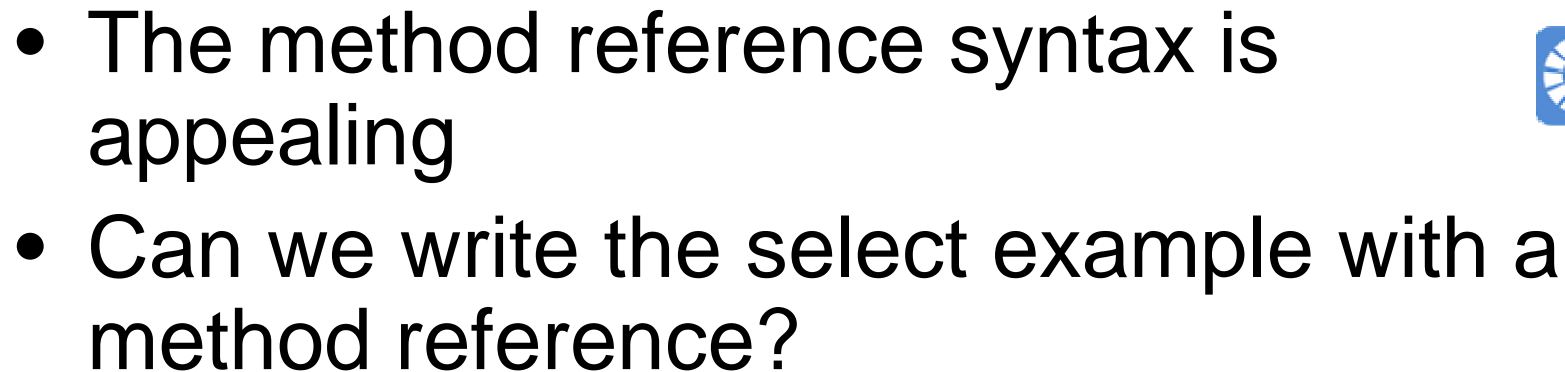
- We upgraded the Kata (our training materials) from Java 7 to Java 8
- Some anonymous inner classes converted easily into Method References

```
MutableList<String> customerCities =
  customers.collect(Customer::getCity);
```

- Some we kept as lambdas

```
MutableList<Customer> customersFromLondon =
  customers.select(customer -> customer.livesIn("London"));
```

Lambdas and Method References



- The method reference syntax is appealing
- Can we write the select example with a method reference?

```
MutableList<Customer> customersFromLondon =
  customers.select(Customer::livesInLondon);
```

- No one writes methods like this.

Lambdas and Method References

- Now we use method references
- We used to use constants

```
MutableList<String> customerCities =
  customers.collect(Customer.TO_CITY);
```

```
public static final Function<Customer, String> TO_CITY =
  new Function<Customer, String>() {
    public String valueOf(Customer customer) {
      return customer.getCity();
    }
  };
```

Lambdas and Method References

- The select example would have created garbage

```
MutableList<Customer> customersFromLondon =
  customers.select(new Predicate<Customer>()
  {
    public boolean accept(Customer customer)
    {
      return customer.livesIn("London");
    }
  });
```

Lambdas and Method References

- So we created `selectWith(Predicate2)` to avoid garbage

```
MutableList<Customer> customersFromLondon =
  customers.selectWith(Customer.LIVES_IN, "London");

public static final Predicate2<Customer, String> LIVES_IN =
  new Predicate2<Customer, String>()
  {
    public boolean accept(Customer customer, String city)
    {
      return customer.livesIn(city);
    }
  };
```

Goldman Sachs

Lambdas and Method References

- The *With() methods work perfectly with Method References

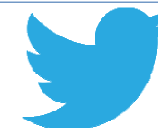
```
MutableList<Customer> customersFromLondon =
  customers.selectWith(Customer::livesIn, "London");
```

- This increases the number of places we can use method references.

Framework Comparisons

Features	GS Collections	Java 8	Guava	Trove	Scala
Rich API	✔	✔	✔		✔
Interfaces	Readable, Mutable, Immutable, FixedSize, Lazy	Mutable, Stream	Mutable, Fluent	Mutable	Readable, Mutable, Immutable, Lazy
Optimized Set & Map	✔ (+Bag)			✔	
Immutable Collections	✔		✔		✔
Primitive Collections	✔ (+Bag, +Immutable)			✔	
Multimaps	✔ (+Bag, +SortedBag)		✔ (+Linked)		(Multimap trait)
Bags (Multisets)	✔		✔		
BiMaps	✔		✔		
Iteration Styles	Eager/Lazy, Serial/Parallel	Lazy, Serial/Parallel	Lazy, Serial	Eager, Serial	Eager/Lazy, Serial/Parallel (Lazy Only)

More Information



@motlin
@GoldmanSachs

stackoverflow.com/questions/tagged/gs-collections

github.com/goldmansachs/gs-collections
github.com/goldmansachs/gs-collections/wiki
github.com/goldmansachs/gs-collections-kata

Parallel-lazy Performance: Java 8 vs Scala vs GS Collections
infoq.com/presentations/java-streams-scala-parallel-collections

GS Collections Memory Benchmark
goldmansachs.com/gs-collections/presentations/GSC_Memory_Tests.pdf

we
BUILD

Learn more at GS.com/Engineering

© 2015 Goldman Sachs. This presentation should not be relied upon or considered investment advice. Goldman Sachs does not warrant or guarantee to anyone the accuracy, completeness or efficacy of this presentation, and recipients should not rely on it except at their own risk. This presentation may not be forwarded or disclosed except with this disclaimer intact.