GS Collections

User Reference Guide

Copyright © 2011 Goldman Sachs. All Rights Reserved.

Version 1.2.0 (03/14/2012)

Contact: gs-collections@gs.com

GS Collections

Contents

About GS Collections.....5

Chapter 1: Iteration patterns.....7

Common iteration patterns.....8
Select/Reject pattern.....8
Collect pattern.....10
Short-circuit patterns.....13
ForEach pattern.....15
InjectInto pattern.....16
RichIterable.....17
Lazy iteration.....17
RichIterable methods.....18
Map iteration methods.....24
Creating iterable views of maps.....24
Collecting entries.....24
Finding, testing and putting values.....25

Chapter 2: Collections and containers.....27

Basic collection types.....28
MutableList.....28
MutableSet.....29
MutableMap.....30
MutableBag.....30
Multimap.....31
Creating and converting collections.....32
Protecting collections.....32
Immutable collections.....32
Protective wrappers.....34

Chapter 3: Code blocks.....35

Common code block types.....36
Predicate.....36
Function.....37
Procedure.....37

Chapter 4: Utility GS Collections.....41

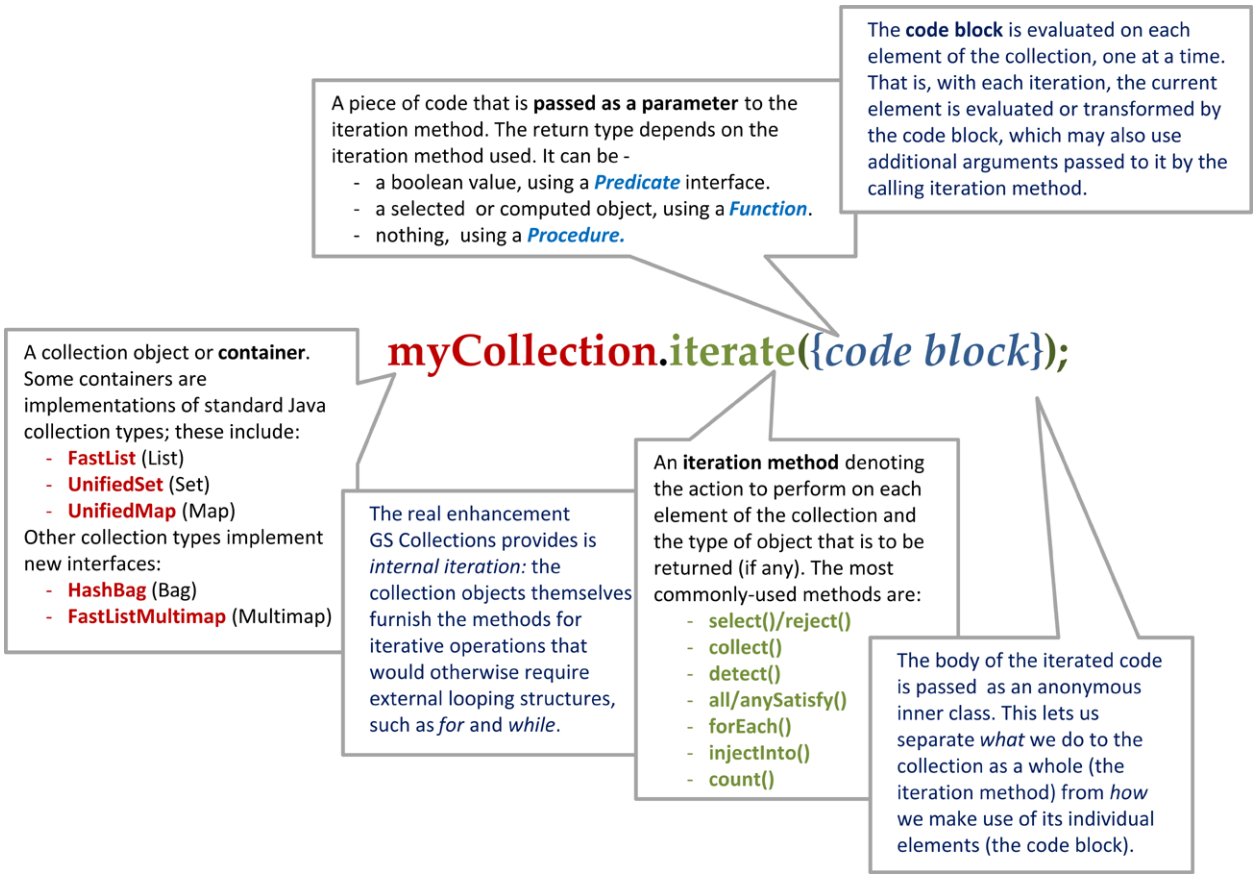
Utility iteration patterns.....42
Parallel iteration.....43

About GS Collections

A collections framework for Java based on Smalltalk patterns.

GS Collections is a library of collection-management utilities that work with the Java Collections Framework (JCF). GS Collections offers JCF-compatible implementation alternatives for List, Set and Map. It also introduces a host of new features including Multimaps and Bags, lazy evaluation, immutable containers, and parallel iteration utility.

GS Collections leverages the idea of *internal iteration* - putting collection-handling methods on the collection classes, a concept derived from the Smalltalk language. Internal iteration affords a more consistent and intuitive approach to using collections by encapsulating the details of how various iteration patterns are implemented. Hiding this complexity lets you write more readable code with less duplication.



About this guide

This Guide is an introduction to basic GS Collections concepts and its commonly-used features. It provides a high-level survey of the GS Collections library and its capabilities. The topics covered are:

- **Iteration patterns**: the logic underlying GS Collections methods.
- **Collections & containers**: the JCF compatible collection types and new containers introduced in GS Collections.
- **Code blocks**: a function that can be passed around as data.
- **Utilities:** Factory classes, static methods, and other features of the library.

Chapter 1

Iteration patterns

Topics:

- *Common iteration patterns*
- *RichIterable*
- *Map iteration methods*

GS Collections extends the Java Collections Framework with new interfaces and classes and provides additional methods. These new methods implement *iteration patterns* derived from the collection protocol of the Smalltalk language (e.g., *select*, *reject*, *collect*, *detect*).

In idiomatic Java, iteration is performed by external **for** and **while** loops that enclose a block of business logic to be performed on each element of a collection. This is an approach that often leads to much duplication of code structure.

In GS Collections, business logic is reified as a *code block*: a class that is passed as a parameter to an iteration method. Each implementation of an iteration method iterates over the collection, passing each element to the code block for processing.

The most important advantage of internal iteration patterns is that they increase readability by giving names to the structural iteration patterns and by reducing code duplication. Moreover, by encapsulating implementation within specialized collection types (e.g., list, sets, maps), iteration can be optimized for the particular type.

Common iteration patterns

The most commonly-used iteration patterns in GS Collections are:

- Filtering patterns:
  - Select
  - Reject
  - Partition
- Transforming patterns:
  - Collect
  - Flatten
  - GroupBy
- "Short-circuit" patterns:
  - Detect
  - AnySatisfy
  - AllSatisfy
- Generic action patterns:
  - ForEach
  - InjectInto

Select/Reject pattern

*Filter a collection to create a new collection: includes **select**, **reject**, and **partition**.*

Methods using the *Select* pattern return a new collection comprising those elements from the source collection that satisfy some logical condition; *Reject* is the inverse pattern, returning a new collection of elements that do *not* satisfy the condition. The condition is a boolean expression in the form of single-argument code block that implements the ***Predicate*** interface.

Select pattern examples:

Pseudocode

```
create <newcollection>
 for each <element> of <collection>
     if condition(<element>)
         add <element> to <newcollection>
```

JDK

```
List<Integer> greaterThanFifty = new ArrayList<Integer>();
for (Integer each : list)
{
    if (each.intValue() > 50)
    {
        greaterThanFifty.add(each);
    }
}
```

Select, using an anonymous inner class as a code block:

GSC

```
MutableList<Integer> greaterThanFifty =
    list.select(new Predicate<Integer>()
```

```
    {
        public boolean accept(Integer each)
        {
            return each > 50;
        }
    });
```

Here, a ***Predicate*** is created using the **Predicates** factory:

GSC

```
list.select(Predicates.greaterThan(50));
```

Here is an example of GS Collections using the proposed Lambda support that is being built in Java 8:

GSC

```
List<Integer> greaterThanFifty =
    list.select(each - > each > 50);
```

Reject pattern examples:

Pseudocode

```
create <newcollection>
 for each <element> of <collection>
     if not condition(<element>)
         add <element> to <newcollection>
```

JDK

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

GSC

```
list.reject(Predicates.greaterThan(50));
```

Select and Reject methods

These GS Collections methods implement the Select and Reject pattern:

select(*Predicate*)

reject(*Predicate*)

The ***Predicate*** is evaluated for each element of the collection. The selected elements are those where the Predicate returned true (false for rejected). The selected (or rejected) elements are returned in a new collection of the same type.

select(*Predicate*, *targetCollection*)

reject(*Predicate*, *targetCollection*)

Same as the **select()**/**reject()** methods with one argument, but results are added to the specified *targetCollection*.

selectWith(*Predicate2*, *argument*)

rejectWith(*Predicate2*, *argument*)

For each element of the collection, ***Predicate2*** is evaluated with the element as one argument, plus one additional argument; selected elements are returned in a new collection of the same type. See *Reusing a code block* for more information.

selectWith(*Predicate2*, *argument*, *targetCollection*)

rejectWith(*Predicate2*, *argument*, *targetCollection*)

Same as the **selectWith()/rejectWith()** methods, but results are added to the specified *targetCollection*.

Partition pattern

Create two collections using Select and Reject.

The *Partition* pattern allocates each element of a collection into one of two new collections depending on whether the element satisfies the condition expressed by the ***Predicate***. In effect, it combines the *Select and Reject patterns*. The collections are returned in a ***PartitionIterable*** specialized for the type of the source collection. You can retrieve the selected and rejected elements from the ***PartitionIterable***. In this example, the list of people is partitioned into lists of adults and children.

GSC

```
MutableList<Person> people =...
PartitionMutableList<Person> partitionedFolks = people.partition(
    new Predicate<Person>()
    {
        public boolean accept(Person each)
        {
            return each.getAge() >= 18;
        }
    });
MutableList<Person> adults = partitionedFolks.getSelected();
MutableList<Person> children = partitionedFolks.getRejected();
```

Partitioning Methods

partition(*Predicate*)

Filters the collection into two separate new collections (selected and rejected), based on ***Predicate***. The collections are returned in a specialized ***PartitionIterable*** of the same type as the source collection.

Collect pattern

*Transform a collection's elements, creating a new collection: includes **collect**, **flatCollect**, and **groupBy**.*

The *Collect* pattern methods return a new collection whose data elements are the results of an evaluation performed by the code block; that is, each element of the original collection is mapped to a new object, which is usually a different type. The code block used as the **collect** method's parameter implements the ***Function*** interface.

Pseudocode

```
create <newcollection>
 for each <element> of <collection>
     <result> = transform(<element>)
     add <result> to <newcollection>
```

JDK

```
List<Address> addresses = new ArrayList<Address>();

for (Person person : people)
{
    addresses.add(person.getAddress());
}
```

GSC

```
MutableList<Person> people =...;

Function<Person, Address> addressFunction =
    new Function<Person, Address>()
    {
        public Address valueOf(Person person)
        {
```

```
            return person.getAddress();
        }
    };

MutableList<Address> addresses = people.collect(addressFunction);
```

Notice that this assumes each person in the **people** collection has just one address. If, instead, a person has multiple addresses, the ***Function*** returns a list of addresses for each person (a list that has only one element if the person has just one address); the result is a List of Lists:

GSC

```
MutableList<Person> people =...;

Function<Person, MutableList<Address>> addressFunction =
    new Function<Person, MutableList<Address>>()
    {
        public MutableList<Address> valueOf(Person person)
        {
            return person.getAddresses();
        }
    };
MutableList<MutableList<Address>> addresses =
    people.collect(addressFunction);
```

Other Collect-style patterns

GS Collections provides two specialized variations on the Collect pattern: *Flatten* and *GroupBy*. Both patterns, like Collect, use an code block implementing a ***Function*** to access and transform element values and return a collection.

Collect methods

These GS Collections methods implement the Collect pattern:

collect(*Function*)

For each element of the collection, the ***Function*** is evaluated with the current element as the argument; returns a new collection of the same size and the transformed type.

collect(*Function*, *targetCollection*)

Same as **collect**, except that the results are added to the specified *targetCollection*.

collectIf(*Predicate*, *Function*)

Same as **collect**, except that the ***Predicate*** is first evaluated with the element as the argument to filter the collection.

collectIf(*Predicate*, *Function*, *targetCollection*)

Same as **collectIf**, except that the results are added to the specified *targetCollection*.

collectWith(*Predicate2*, *argument2*)

Same as **collect**, but the ***Predicate2*** is evaluated with the element as one argument, plus one additional argument; returns a new collection of the same size and the transformed type.

collectWith(*Predicate2*, *argument2*, *targetCollection*)

Same as **collectWith**, except that the results are added to a specified *targetCollection*.

Flatten pattern

Create a single, linear collection from selected values of a collection's elements.

The *Flatten* pattern is a specialized form of the the Collect pattern. It returns a single-level, or "flattened," collection of attribute values from a source collection's elements.

flatCollect(*Function*)

Extract (and, optionally, transform) a selected attribute value from each of the calling collection's elements and add the values to a new single-level collection (e.g., a List or Set).

Given list of **people** (as in the *Collect* example), here is how **flatCollect** could be used to create a flat list from the address fields of the person objects in that list, using the same ***Function*** (**addressFunction**):

Pseudocode

```
create <newcollection>
 for each <element> of <collection>
     <results> = transform(<element>)
     Add all <results> to <newcollection>
```

JDK

```
List<Address> addresses = new ArrayList<Address>();
for (Person person : people)
{
  addresses.addAll(person.getAddresses());
}
```

GSC

```
MutableList<Address> addresses =  people.flatCollect(addressFunction);
```

Note the **flatCollect** method's similarity to a **collect** method having the same signature: each method's ***Function*** parameter maps to an Iterable type. This is optional for **collect**, but required of **flatCollect.** Both methods return a new collection. The difference is that **collect** in this form creates a *collection of collections* from a simple List, Set or Bag, while **flatCollect** performs a different (and in this instance, somewhat more useful) action, returning a flat list of addresses.

GroupBy pattern

Create a Multimap from a collection by grouping on a selected or generated key value.

The *GroupBy* pattern gathers the elements on the collection into a map-like container called a ***Multimap***, which associates multiple values for each key. The ***Function*** is applied to each element and the result is used as the key into the Multimap. The elements are traversed in the same order as they would be in a *ForEach pattern* .

groupBy(*Function*)

Group the elements into a new ***Multimap***. Uses the ***Function*** to get the key for each element.

groupBy(*Function*, *targetMultimap*)

Same as **groupBy** except that results are added to the specified *targetMultimap*.

groupByEach(*Function*)

Same as **groupBy** except that the ***Function*** transforms each value into multiple keys, returning a new ***Multimap*** containing all the key/value pairs.

Short-circuit patterns

*Methods that control processing by testing a collection for a logical condition: includes **detect**, **anySatisfy**, and **allSatisfy**.*

The "short-circuit" patterns - *Detect*, *AnySatisfy* and *AllSatisfy* - are so called because they describe methods that cease execution when a specific condition is met. With each iteration, the ***Predicate*** is evaluated. If the evaluation resolves as a specified boolean (*true*/*false*) value, then iteration halts and returns the appropriate value.

Detect pattern

Finds and returns the first element that satisfies a given logical expression.

Detect returns the first element that satisfies a ***Predicate***. If no *true* evaluation occurs, all elements in the collection are tested and the method returns*null*.

Pseudocode

```
for each <element> of <collection>
  if condition(<element>)
    return <element>
```

JDK

```
for (int i = 0; i < list.size(); i++)
{
    Integer v = list.get(i);
    if (v.intValue() > 50)
    {
        return v;
    }
    return null;
}
```

GSC

```
list.detect(Predicates.greaterThan(50));
```

Detect methods

detect(*Predicate*)

Return the first element of the collection for which Predicate evaluates as *true* when given that element as an argument; if no element causes ***Predicate*** to evaluate as *true*, the method returns *null*.

detectIfNone(*Predicate*, *Function0*)

Same as **detect**, but if no element causes ***Predicate*** to evaluate as *true*, return the result of evaluating ***Function0***.

AnySatisfy pattern

Determine if any collection element satisfies a given logical expression.

The *AnySatisfy*- method tests for the first occurrence of an element that the ***Predicate*** evaluates as *true*. If such an element is found, execution halts and the method returns *true*; otherwise, the it returns *false*.

Pseudocode

```
for each <element> of <collection>
     if condition(<element>)
         return true
 otherwise return false
```

JDK

```
for (int i = 0; i < list.size(); i++)
{
    Integer v = list.get(i);
    if (v.intValue() > 50)
```

```
    {
        return true;
    }
}
return false;
```

GSC

```
return list.anySatisfy(Predicates.greaterThan(50));
```

AnySatisfy methods

anySatisfy(*Predicate*)

Return true if the ***Predicate*** evaluates as *true* for any element of the collection. Otherwise (or if the collection is empty), return false.

anySatisfyWith(*Predicate2*, *parameter*)

Return true if the ***Predicate2*** evaluates as *true* for any element of the collection. Otherwise (or if the collection is empty), return false.

AllSatisfy pattern

Determine if all collection elements satisfy a given logical expression.

The *AllSatisfy*-pattern method determines whether *all* elements satisfy the ***Predicate***; that is, it seeks the first element that evaluates as *false* for the given predicate. If such as element is found, execution halts and the method returns *false.* Otherwise, the method returns *true*.

Pseudocode

```
for each <element> of <collection>
     if not condition(<element>)
         return false
 otherwise return true
```

JDK

```
for (int i = 0; i < list.size(); i++)
{
    Integer v = list.get(i);
    if (v.intValue() <= 50)
    {
        return false;
    }
}
return true;
```

GSC

```
return list.allSatisfy(Predicates.greaterThan(50));
```

AllSatisfy methods

allSatisfy(*Predicate*)

Return true if the ***Predicate*** evaluates as *true* for all elements of the collection. Otherwise (or if the collection is empty), return *false*.

allSatisfyWith(*Predicate2*, *parameter*)

Return true if the ***Predicate2*** evaluates as *true* for all elements of the collection. Otherwise (or if the collection is empty), return *false*.

ForEach pattern

Perform a calculation on each element of the current collection.

The *ForEach* pattern defines the most basic iteration operation that can be used with all collection types. Unlike the other patterns discussed in this topic, the ForEach pattern prescribes methods that operate on each element of the calling collection object, with no value returned by the method itself.

In GS Collections, the **forEach** method offers the most straightforward replacement for the Java for loop. It executes the code in a Procedure on each element. You can use these methods to perform some action using the values of the source collection - for example, to print a value or to call another method on each element.

Pseudocode

```
for each <element> of <collection>
     evaluate(<element>)
```

JDK

```
for (int i = 0; i < list.size(); i++)
{
    this.doSomething(list.get(i));
}
```

GSC

```
list.forEach(new Procedure()
{
    public void value(Object each)
    {
        doSomething(each);
    }
});
```

ForEach methods

forEach(*Procedure*)

For each element, the code block is evaluated with the element as the argument.

forEachIf(*Predicate*, *Procedure*)

For each element where ***Predicate*** evaluates as true, ***Procedure*** is evaluated with the current element as the argument.

forEach(*Procedure*, *fromIndex*, *toindex*)

Iterates over the section of a ***MutableList*** covered by the specified inclusive indexes.

forEachWith(*Procedure2*, *parameter*)

For each element of the collection, the code block is evaluated with the element as the first argument, and the specified *parameter* as the second argument.

forEachWithIndex(*ObjectIntProcedure*)

Iterates over a collection passing each element and the current relative int index to the specified instance of ProcedureWithInt

forEachWithIndex(*ObjectIntProcedure*, *fromIndex*, *toIndex*)

Iterates over the section of the list covered by the specified inclusive indexes.

InjectInto pattern

Calculate and maintain a running value during iteration; use each evaluated result as an argument in the next iteration.

The *InjectInto* pattern is used to carry a computed result from one iteration as input to the next. In this pattern, the **injectInto** method takes an initial *injected value* as a parameter. This value is used as the first argument to a two-argument code block; the current element (for each iteration of the collection) is taken as the second argument.

For each iteration, the code block's evaluation result is passed to the next iteration as the first argument (the injected value) of the code block, with the (new) current element as the second argument. The injectInto() method returns the code block's cumulative result upon the final iteration.

Pseudocode

```
set <result> to <initialvalue>
 for each <element> of <collection>
     <result> = apply(<result>, <element>)
 return <result>
```

JDK

```
List<Integer> list = Lists.mutable.of(1, 2);
int result = 5;
for (int i = 0; i < list.size(); i++)
{
    Integer v = list.get(i);
    result = result + v.intValue();
}
```

GSC

```
Lists.mutable.of(1, 2).injectInto(3, AddFunction.INTEGER);
```

InjectInto methods

injectInto(*injectedValue*, ***Function2***)

Return the final result of all evaluations using as the arguments each element of the collection, and the result of the previous iteration's evaluation.

injectInto(*intInjectedValue*, ***IntObjectToIntFunction***)

Return the final result of all evaluations using as the arguments each element of the collection, and the result of the previous iteration's evaluation.

injectInto(*intValue*, ***IntObjectToIntFunction***)

Return the final result of all evaluations using as the arguments each element of the collection, and the result of the previous iteration's evaluation. The injected value and final result are both primitive ints.

injectInto(*longValue*, ***LongObjectToLongFunction***)

Return the final result of all evaluations using as the arguments each element of the collection, and the result of the previous iteration's evaluation. The injected value and result are both primitive longs.

injectInto(*doubleValue*, ***DoubleObjectToDoubleFunction***)

Return the final result of all evaluations using as the arguments each element of the collection, and the result of the previous iteration's evaluation. The injected value and result are both primitive doubles.

RichIterable

RichIterable is the most important interface in GS Collections. It provides the blueprint for all non-mutating iteration patterns. It represents an object made up of elements that can be individually and consecutively viewed or evaluated (an *iterable*), and it prescribes the actions that can be performed with each evaluation (the patterns). The most commonly used implementations include **FastList** and **UnifiedSet**.

RichIterable is extended by ***ListIterable***, ***SetIterable***, ***Bag***, and ***MapIterable***. ***MapIterable*** is iterable on its values using the ***RichIterable*** API.

RichIterable is also extended by ***MutableCollection***, and indirectly by ***MutableList*** and ***MutableSet*** (which also extend the mutable Java Collection types List and Set). Another subinterface defines a non-JDK container, ***MutableBag*** (or multiset); yet another, ***ImmutableCollection***, delineates the immutable forms of these GS Collections containers. These latter two interfaces are detailed in the next topic.

The subinterface ***LazyIterable*** for the most part replicates ***RichIterable***, but overrides some specific collection-returning methods - **collect**, **collectIf**, **select**, **reject**, and **flatCollect** - so that they delay their actual execution until the returned collection is needed, a technique called "lazy iteration."

Lazy iteration

Deferring evaluation until necessary.

Lazy iteration is an optimization pattern in which an iteration method is invoked, but its actual execution is deferred until its action or return values are required by another, subsequent method. In practical terms, the objective is typically to forestall unnecessary processing, memory use, and temporary-object creation unless and until they are needed. Lazy iteration is implemented as an adapter on the current *RichIterable* collection by this method:

richIterable.**asLazy()**	Returns a deferred-evaluation iterable. (Note the list below of other GS Collections methods that return lazy Iterables.)

In a way, lazy iteration is a companion to the short-circuit iteration pattern described earlier, in which iteration ceases as soon the method's purpose is achieved. In the last line of the example below, the anySatisfy() method quits execution when it detects the "address2" element in the **addresses** list created by **collect()**. The third element ("address 3") is never examined by **anySatisfy** - although it was present in **addresses**.

GSC

```
Person person1 = new Person(address1);
Person person2 = new Person(address2);
Person person3 = new Person(address3);
MutableList<Person> people =
    FastList.newListWith(person1, person2, person3);
MutableList<MutableList<Address>> addresses =
    people.collect(addressFunction);
Assert.assertTrue(addresses.anySatisfy(Predicates.equal(address2)));
```

One excess element out of three may be trivial, but if people were to be very long list (or a stream), **anySatisfy** will still have to wait for the **collect** method to finish aggregating an equally-large temporary collection - one that will only have its first two elements inspected. By applying a lazy-iteration adapter to **people**, the **collect** iteration defers to that of **anySatisfy**: only the elements **anySatisfy** requires are "collected."

GSC

```
MutableList<Person> people = FastList.newListWith(person1, person2, person3);
LazyIterable<Person> lazyPeople = people.asLazy();
LazyIterable<Address> addresses = lazyPeople.collect(addressFunction);
Assert.assertTrue(addresses.anySatisfy(Predicates.equal(address2)));
```

In this example, the values in a Multimap are flattened and sorted, the results processed and sent to a stream by **forEach**.

GSC

```
myMultimap.multiValuesView()       // returns a lazy iterable by default
    .select(ITERABLE_SIZE_AT_THRESHOLD)      // invoked but deferred…
    .asSortedList(DESCENDING_ITERABLE_SIZE)  // as "select" evaluates,
                                             // sort elements in a non-lazy
                                             // sorted list.
    .asLazy()                                // restores the lazy adapter
    .collect(ITERABLE_TO_FORMATTED_STRING)   // invoked but deferred…
    .forEach(Procedures.println(System.out));// as "collect" evaluates,
                                             // send results to stream.
```

Because a lazy iterable adapter is used, the *collect* evaluation occurs only as the *forEach* evaluation calls for it; there is no intervening collection. Without the lazy adapter, **collect()** would execute in full, then return a collection to **forEach**.

Finally, note these GS Collections methods that implicitly return a lazy-iterable type.

MutableMap interface and its implementations

valuesView()	An unmodifiable view of the map's values.
keysView()	An unmodifiable view of the map's keyset.
entriesView()	An unmodifiable view of the map's entryset.

Multimap interface and its implementations

keyMultiValuePairsView()	An unmodifiable view of key and multi-value pairs.
keysView()	An unmodifiable view of unique keys.
keyValuePairsView()	An unmodifiable view of key/value pairs.
multiValuesView()	An unmodifiable view of each key's values, without the key.

RichIterable methods

These methods are available on all implementations of ***RichIterable***.

Building strings

Methods that convert collection elements to a string that can be appended to a stream or buffer.

The **makeString** method returns a representation of the calling ***RichIterable*** collection as a **String** object. Elements are converted to strings as they would be by **String.valueOf(Object)**. You can specify start and end strings as delimiters (the default is an empty string for both) and the separator string for the between-values delimiter (defaults to comma and space).

makeString(*startString*, *separatorString*, *endString***)**	Returns a string representation of the calling collection that is a list of elements in the order they are returned by the iterator, enclosed in the startString and endString. Elements are delimited by the separatorString.
makeString(*separatorString***)**	Same result with no starting and ending strings.

makeString()	Same result with the default delimiter ", " (comma space) and no starting and ending strings.

GSC

```
MutableList<Integer> list = FastList.newListWith(1, 2, 3);
String myDelim = list.makeString("[", "/", "]"); // "[1/2/3]"
String mySeper = list.makeString("/");           // "1/2/3"
String
default=list.makeString();                       //"1, 2, 3"
```

The **appendString** method uses forms similar to **makeString**, but the string representation of the collection is written to a Java Appendable object, such as a PrintStream, StringBuilder or StringBuffer; the method itself is void.

appendString(Appendable, *startString*, *separatorString*, *endString***)**	Appends a string representation of this collection onto the given Appendable using the specified start, end, and separator strings
appendString(Appendable, *separatorString***)**	Appends with specified separator, but no starting or ending strings.
appendString(Appendable**)**	Appends with the default delimiter ", " (comma space) and no starting and ending strings.

GSC

```
MutableList<Integer> list = FastList.newListWith(1, 2, 3);
Appendable myStringBuider = new StringBuilder();
list.appendString(myStringBuider, "[", "/", "]"); //"[1/2/3]");
```

Counting elements

Get the total number of elements that satisfy a condition.

The **count** and **countWith** methods calculate the number of collection elements that satisfy a given predicate. The **countWith** method takes a second parameter that is used as an additional argument in evaluating the current element.

count(*Predicate*)

For each element of the collection, ***Predicate*** is evaluated with the current element as its argument. The count is incremented if the ***Predicate*** evaluates as *true*. For example:

GSC

```
return people.count(new Predicate<Person>() {
public boolean value(Person person) {
  return person.getAddress().getState().getName().equals("New York");
}
});
```

countWith(*Predicate2*, *parameter*)

For each element of the collection, ***Predicate2*** is evaluated with the element as the first argument and the specified parameter as the second argument. The count is incremented if the discriminator evaluates as *true*.

GSC

```
return lastNames.countWith(Predicate2.equal(), "Smith");
```

Use these methods to get the total number of collection items or to determine whether the collection is empty.

size()	Returns the number of items in the collection.

isEmpty()	Returns *true* if this iterable has zero items.
notEmpty()	Returns *true* if this iterable has greater than zero items.

Finding elements

Locate elements by iteration position or highest/lowest value.

The **getFirst** and **getLast** methods return the first and last elements, respectively of a ***RichIterable*** collection. In the case of a List, these are the elements at the first and last index. For all any other collections, **getFirst** and **getLast** return the first and last elements that would be returned by an iterator. Note that the first or last element of a hash-based Set could be any element, because element order in a hashed structure is not defined. Both methods return *null* if the collection is empty. If null is a valid element, use the **isEmpty** method to determine if the container is in fact empty.

getFirst()	Returns the first element of an iterable collection.
getLast()	Returns the last element of an iterable collection.

The **min()** and max() methods, without parameters, return an element from an iterable based on its natural order, that is, by calling the compareTo() method on each element.

max()	Returns the maximum value out of a collection of Comparable objects (e.g., **List**<Integer>).
min()	Returns the minimum value out of a collection of Comparable objects (e.g. **List**<Integer>).

GSC

```
RichIterable<Integer> iterable = FastList.newListWith(5, 4, 8, 9, 1);
Assert.assertEquals(Integer.valueOf(9), iterable.max());
Assert.assertEquals(Integer.valueOf(1), iterable.min());
```

If any element in the iterable is not comparable, then a ClassCastException is thrown.

GSC

```
RichIterable<Object> iterable = FastList.newListWith(5, 4, 8, 9, 1, new Foo());
iterable.max(); // throws ClassCastException
```

The **min()** and **max()** methods each have an overload that takes a Comparator that determines the natural order.

max(*Comparator***)**	Returns the maximum element out of this collection based on the comparator.
min(*Comparator***)**	Returns the minimum element out of this collection based on the comparator.

GSC

```
public class SillyWalk
{
    public final int wiggles;

    public SillyWalk(int wiggles)
    {
        this.wiggles = wiggles;
    }
}
```

```
private static final Comparator<SillyWalk> SILLY_WALK_COMPARATOR =
    new Comparator<SillyWalk>()
    {
        public int compare(SillyWalk o1, SillyWalk o2)
        {
            return o1.wiggles - o2.wiggles;
        }
    };
```

```
SillyWalk sillyWalk2 = new SillyWalk(2);
SillyWalk sillyWalk3 = new SillyWalk(3);

RichIterable<SillyWalk> walks = FastList.newListWith(sillyWalk2, sillyWalk3);

Assert.assertEquals(sillyWalk3, walks.max(SILLY_WALK_COMPARATOR));
Assert.assertEquals(sillyWalk2, walks.min(SILLY_WALK_COMPARATOR));
```

The related methods **minBy()** and **maxBy()** take a ***Function*** and return the minimum or maximum element in the ***RichIterable*** based on the natural order of the attribute returned by the selector.

maxBy(*Function*)	Returns the maximum element out of this collection based on the result of applying the ***Function*** to each element.
minBy(*Function*)	Returns the minimum element out of this collection based on the result of applying the ***Function*** to each element.

Here, we find the youngest person (the minimum person by age).

GSC

```
Person alice = new Person("Alice", 40);
Person bob = new Person("Bob", 30);
Person charlie = new Person("Charlie", 50);

MutableList<Person> people = FastList.newListWith(alice, bob, charlie);

Assert.assertEquals(bob, people.minBy(Person.TO_AGE));
```

In the code example we already had an Function, so calling **minBy()** was more concise than calling **min()**. These two forms are equivalent though.

GSC

```
people.minBy(Person.TO_AGE);
people.min(Comparators.byFunction(Person.TO_AGE));
```

Using chunk and zip to create collections

Grouping and pairing elements of one or more collections.

The **chunk** method can be used to gather the elements of a collection into *chunks*; that is, it creates a collection made up of collections of a specified fixed *size* (an integer). If the *size* doesn't divide evenly into the total of collection elements, then the final chunk is smaller.

chunk(*size***)**

Returns a new collection with the source collection's elements grouped in "chunks," with *size* elements in each chunk, and the last chunk containing the remaining elements, if any.

GSC

```
MutableList<Integer> list =
    FastList.newListWith(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
RichIterable<RichIterable<Integer>> chunks = list.chunk(4);

System.out.println(chunks);
```

This example prints out:

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

The **zip** method pairs up the elements of one ***RichIterable*** with those of second. If one of the two collections has more elements than the other, those remaining elements are dropped. The **zipWithIndex** method is a special case of **zip** that pairs the elements in a collection with their index positions.

zip(*RichIterable*)

Returns a new ***RichIterable*** by combining, into pairs, corresponding elements from the calling RichIterable collection and the RichIterable collection named in the parameter. If one of the two collections is longer, its remaining elements are ignored..

GSC

```
MutableList<String> list1 =
 FastList.newListWith("One", "Two", "Three", "Truncated");
MutableList<String> list2 = FastList.newListWith("Four", "Five", "Six");
MutableList<Pair<String, String>> pairs = list1.zip(list2);
System.out.println(pairs);
```

This example prints out:

```
[One:Four, Two:Five, Three:Six]
```

zipWithIndex()

Returns a new ***RichIterable*** consisting of the calling collection's elements, each paired with its index (beginning with index 0).

GSC

```
MutableList<String> list = FastList.newListWith("One", "Two", "Three");
MutableList<Pair<String, Integer>> pairs = list.zipWithIndex();
System.out.println(pairs);
```

This example prints out:

```
[One:0, Two:1, Three:2]
```

Performance optimized methods: reusing two-argument code blocks

*Using **selectWith**, **rejectWith**, and **collectWith** inside other iteration patterns (or loops) where code blocks can be created outside of the outer iteration patterns or made static.*

The iteration patterns **collect**, **select**, and **reject** each take a single parameter, a code block that itself takes a single argument. These patterns have alternate forms, methods named **collectWith**, **selectWith**, and **rejectWith** respectively, which take two parameters:

- The first method parameter is a code block that itself takes *two* arguments; the first argument of the code block is the current element with each iteration.
- The second method parameter is an object that is then passed to the code block as its second argument.

selectWith(*Predicate2*, *argument*)

rejectWith(*Predicate2*, *argument*)

For each element of the collection, ***Predicate2*** is evaluated with the element as one argument, plus one additional argument; selected or rejected elements are returned in a new collection of the same type.

collectWith(*Predicate2*, *argument*)

Same as the **collect** method, but two arguments are passed to the code block; returns a new collection of the same type and size.

These "- **With**" forms accomplish exactly the same actions as their basic counterparts. Although slightly more verbose, they allow for a specific performance optimization, that is re-use of the code block with different arguments. Here is an example of **select** that finds the adults in a list of people. First, the JDK version, and then rewritten in GS Collections form:

JDK

```
List<Person> people =...;
List<Person> adults = new ArrayList<Person>();
for (Person person : people)
{
  if (person.getAge() >= 18)
  {
      adults.add(person);
  }
}
```

GSC

```
MutableList<Person> people =...;
MutableList<Person> adults = people.select(
  new Predicate<Person>()
  {
      public boolean accept(Person each)
      {
          return each.getAge() >= 18;
      }
  });
```

Here's the same algorithm, again in GS Collections, this time using **selectWith()**:

GSC

```
MutableList<Person> people =...;
MutableList<Person> adults = people.selectWith(
  new Predicate2<Person, Integer>()
  {
      @Override
      public boolean accept(Person eachPerson, Integer age)
      {
          return eachPerson.getAge() > age;
      }
  }, 18);
```

In this single instance, there is no reason to write it out this longer way; the extra generality - making *age* the second argument to the ***Predicate2*** - is unnecessary.

It does make sense, however, if you wanted to filter on multiple ages: you could hold onto and re-use the ***Predicate2***, thereby creating less garbage.

GSC

```
MutableList<Person> people =...;
Predicate2<Person, Integer> agePredicate =
  new Predicate2<Person, Integer>()
  {
      @Override
      public boolean accept(Person eachPerson, Integer age)
      {
          return eachPerson.getAge() > age;
      }
  };
MutableList<Person> drivers = people.selectWith(agePredicate, 17);
MutableList<Person> voters = people.selectWith(agePredicate, 18);
MutableList<Person> drinkers = people.selectWith(agePredicate, 21);
```

Map iteration methods

Containers derived from the Map interfaces (***MapIterable***, ***MutableMap***, ***ImmutableMap***) and the Multimap interfaces (***MutableListMultimap***, et al.) differ from those implementing ***MutableList***, ***MutableSet***, and ***MutableBag***, all of whose iteration patterns are specified by ***RichIterable***. Maps, are a special case, comprising two separate, though joined groups of elements: a set of *keys* and their associated *values*.

To enable iteration over the special structure of Maps and Multimaps, GS Collections provides a set of map-specific methods whose operations and returned objects are specific to the keys, values, and entries (combined key-value elements) that make up a Map.

Creating iterable views of maps

Wrapper classes that return an iterable view of a map; ForEach patterns for Map containers.

These three methods each return an unmodifiable RichIterable view of a Map. They are essentially wrappers over the modifiable, non-lazy objects returned by the corresponding Java Collections Framework methods.

valuesView()	*(Maps and Multimaps)* Returns an unmodifiable ***RichIterable*** wrapper over the values of the Map.
keysView()	*(Maps and Multimaps)* Returns an unmodifiable ***RichIterable*** wrapper over the keySet of the Map.
entriesView()	*(Maps only)* Returns an unmodifiable ***RichIterable*** wrapper over the entrySet of the Map.

ForEach Iteration

These three methods call a code block for each element on a Map (all return void).

forEachKey(*Procedure*)	Calls the ***Procedure*** on each key of the Map.
forEachValue(*Procedure*)	Calls the ***Procedure*** on each value of the Map.
forEachKeyValue(*Procedure2*)	Calls the ***Procedure*** on each key-value pair of the Map.

Collecting entries

Gather entries from another collection into a Map

Use the **collectKeysAndValues** method to add all the entries derived from another collection into the current Map.

collectKeysAndValues(*collection*, *keySelector*, *valueSelector*)

(Maps only) The key and value for each entry is determined by applying the *keySelector* and *valueSelector* (in each case, a ***Function***) to each item in *collection*. Each is converted into a key-value entry and inserted into the Map. If a new entry has the same key as an existing entry in the calling map, the new entry's value replaces that of the existing entry.

Finding, testing and putting values

Detect a value by its key and, optionally, insert or return other values.

The **getIfAbsent…** and **ifPresentApply** methods locate a specified key and return a map value that corresponds to that key. Depending on whether a value is found at the given key, each method performs a specific action.

---

getIfAbsent(*key*, ***Function0***)

Returns the value in the Map that corresponds to the specified *key*; if there is no value at the key, returns the result of evaluating the specified ***Function0*** (here, specifically, a code block without parameters that returns some object).

---

getIfAbsentPut(*key*, ***Function0***)

Returns the value in the Map that corresponds to the specified *key*; if there is no value at the key, returns the result of evaluating the specified ***Function0***, and puts that value in the map at the specified key

---

getIfAbsentPutWith(*key*, ***Function***, *parameter***)**

Returns the value in the Map that corresponds to the specified *key*; if there is no value at the key, returns the result of evaluating the specified one-argument Function using the specified *parameter*, and put that value in the map at the specified key.

---

getIfAbsentWith(*key*, ***Function***, *parameter***)**

Returns the value in the Map that corresponds to the specified *key*; if there is no value at the key, returns the result of evaluating the specified ***Function*** and parameter.

---

ifPresentApply(*key*, ***Function***)

If there is a value in the Map that corresponds to the specified *key*, returns the result of evaluating the specified ***Function*** with the value, otherwise returns null.

Chapter 2

Collections and containers

Topics:

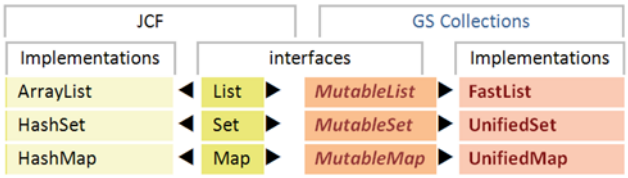
- *Basic collection types*
- *Creating and converting collections*
- *Protecting collections*

What is perhaps most distinctive about the GS Collections collection classes is what (quite properly) is hidden: their implementation of iteration patterns. Through this encapsulation, GS Collections is able to provide optimized versions of each method on each container. For example, the first of the classes we'll discuss here, **FastList** is array-based; it iterates using indexed access directly against its internal array.

We'll begin with the GS Collections implementations of types having analogs in the Java Collections Framework (JCF). We'll then discuss the new types ***Bag*** and ***Multimap***, the Immutable collections, and protective wrappers.

Basic collection types

The most commonly-used GS Collections classes are **FastList**, **UnifiedSet**, and **UnifiedMap**. These collections serve as drop-in replacements for their corresponding types in the Java Collections Framework (JCF). Note that these GS Collections classes do not extend the JCF implementations; they are instead new implementations of both JCF and GS Collections interfaces, as this (highly-simplified) diagram summarizes:



The methods of the JCF types are primarily focused on adding or removing elements and similar, non-iterative operations. GS Collections interfaces provide methods for iteration patterns that for the most part, do not modify (mutate) the source collection, but rather return a new collection or information about the source collection.

MutableList

An ordered collection that allows duplicate elements.

The ***MutableList*** interface (extending ***ListIterable***) describes a collection of elements that have a specific order, with duplicate values permitted.

MutableList extends the JCF **List** interface and has the same contract. It also extends ***RichIterable*** which provides the iteration methods described in the previous topic,

The most common implementation of ***MutableList*** is **FastList**, which can be used to replace the familiar java.util.ArrayList.Here is a comparison of how the two types can be created.

Class	Example
ArrayList (JCF)	`List<String> comparison = new ArrayList<String>(); comparison.add("Comcast"); comparison.add("IBM); comparison.add("Microsoft"); comparison.add("Microsoft"); return comparison;`
FastList (GSC)	`return FastList.newListWith("Comcast","IBM", "Microsoft", "Microsoft");`

The ***MutableList*** interface includes the **sortThis** and **reverse** methods, which are similar to the static methods with the same names on java.util.Collections.Both are mutating methods. Here is an example of using **sort** using the JDK API and then GS Collections

Class	Example
ArrayList (JCF)	`Collections.sort(people, new Comparator<Person>() { public int compare(Person o1, Person o2) { int lastName = o1.getLastName().compareTo(o2.getLastName()); if (lastName != 0) { return lastName; } return o1.getFirstName().compareTo(o2.getFirstName()); }`

Class	Example
	`});`

FastList (GSC)	`people.sortThis(new Comparator<Person>()`
	`{`
	`  public int compare(Person o1, Person o2)`
	`  {`
	`  int lastName = o1.getLastName().compareTo(o2.getLastName());`
	`  if (lastName != 0)`
	`  {`
	`  return lastName;`
	`  }`
	`  return o1.getFirstName().compareTo(o2.getFirstName());`
	`  }`
	`});`

MutableList adds a new method called **sortThisBy**, which gets some attribute from each element using a ***Function*** and then sorts the list by the natural order of that attribute.

Class	Example
ArrayList (JCF)	`Collections.sort(people, Functions.toComparator(Person.TO_AGE));`
FastList (GSC)	`people.sortThisBy(Person.TO_AGE);`

Here is an example comparing reverse() using the JCF and using GS Collections; both are mutating methods.

Class	Example
ArrayList (JCF)	`Collections.reverse(people);`
FastList (GSC)	`people.reverse();`

The **toReversed** method on ***MutableList*** lets you reverse a list *without* mutating it. Here is an example of how to accomplish that in the JCF and in GS Collections.

Class	Example
ArrayList (JCF)	`List<Person> reversed = new ArrayList<Person>(people)` `Collections.reverse(reversed);`
FastList (GSC)	`MutableList<Person> reversed = people.toReversed();`

MutableSet

An unordered collection that allows no duplicate elements.

The ***MutableSet*** interface (extending ***SetIterable***) defines an unordered collection that does not permit duplicate elements. An attempt to add duplicate elements to a ***MutableSet*** container is ignored without throwing an exception. The order in which the elements are processed during iteration is not specified.

MutableSet extends SetIterable and has the same contract. The most common implementation is **UnifiedSet**, which is the GS Collections counterpart of HashSet in the Java Collections Framework. As with ***MutableList***, the ***MutableSet*** interface extends the ***RichIterable*** interface.

Class	Example

HashSet (JDK)	`Set<String> comparison = new HashSet<String>();`
	`comparison.add("IBM");`
	`comparison.add("Microsoft");`
	`comparison.add("Oracle");`

Class	Example
	`comparison.add("Comcast");` `return comparison;`
UnifiedSet (GSC)	`return UnifiedSet.newSetWith("IBM", "Microsoft", "Verizon", "Comcast");`

MutableMap

A collection of key/value pairs

The ***MutableMap*** interface defines an association of key/value pairs. It extends the ***MapIterable*** interface, which furnishes a set of iteration methods especially for the key/value structure of a Map collection. These include unmodifiable views of keys, values or pair-entries using the **keysView**, **valuesView** and **entriesView** methods, respectively.

The mutable subinterfaces of MapIterable also extend the JCF Map interface.

The most common implementation of ***MutableMap*** is **UnifiedMap**, which can replace the Java class HashMap.

Class	Example

HashMap (JDK)	`Map<Integer, String> map = new HashMap<Integer, String>();`
	`map.put(1, "1");`
	`map.put(2, "2");`
	`map.put(3, "3");`

UnifiedMap (GSC)	`MutableMap<Integer, String> map = UnifiedMap.newWithKeysValues(1, "1", 2, "2", 3, "3");`

MutableBag

An unordered collection that allows duplicates.

A ***MutableBag*** (extending ***Bag***) combines the less-restrictive aspects of a Set - in that it is an unordered collection - and a List, which permits adding duplicate values. It is implemented using a specialized kind of map, called a *multiset*, which pairs each distinct value *as a key* with the count of its occurrences in the collection *as a value*.

For example, this list:

```
Apple
Pear
Orange
Orange
Apple
Orange
```

could create this bag:

Pear	1
Orange	3
Apple	2

GSC	`return MutableBag < String > bag =` `HashBag.newBagWith("Apple", "Pear", "Orange", "Apple", "Apple", "Orange");`

The ***MutableBag*** interface includes methods for getting and manipulating the number of occurrences of an item. For example, to determine the number of unique elements in a MutableBag, use the **sizeDistinct()** method.

Multimap

A map-like container that can have multiple values for each key

In a ***Multimap*** container, each key can be associated with multiple values. It is, in this sense, similar to a Map, but one whose values consist of individual collections of a specified type, called the *backing collection*. A ***Multimap*** is useful in situations where you would otherwise use ***Map***<K, Collection<V>>.

Unlike the other basic GS Collections containers, ***Multimap*** does not extend ***RichIterable***, but resides along with its subinterfaces in a separate API. The ***RichIterable*** methods are extended by the backing collection types.

Depending on the implementation, the "values" in a Multimap can be stored in Lists, Sets or Bags. For example, the **FastListMultimap** class is backed by a **UnifiedMap** that associates each key with a **FastList** that preserves the order in which the values are added and allows duplicate to be added.

A ***Multimap*** is the type returned by the ***groupBy*** method. Here is an example in which we group a list of words by their length, obtaining a ***Multimap*** with integer (word=length) keys and lists of words having that length for values.

This simple list:

```
here
are
a
few
words
that
are
not
too
long
```

produces a List-backed ***Multimap***:

key	value<list>
1	a
3	are,few,are,not,too
4	here,that,long
5	words

The code that performs this action uses the ***groupBy*** method.

GSC

```
MutableList<String> words = FastList.newListWith("here", "are", "a", "few",
  "words", "that", "are", "not", "too", "long");
MutableListMultimap<Integer, String> multimap =
 words.groupBy(StringFunctions.length());
```

The interface ***MutableListMultimap*** extends the ***Multimap*** interface and tells us the type of its backing collections. Since this example uses Lists, the word "are" is allowed to occur twice in the list at key 3.

If we use **groupBy** on the same source list to generate a ***Multimap*** of Sets, the resulting backing collections will eliminate duplicate entries and disregard the order of elements in the source List:

GSC

```
MutableSetMultimap<Integer, String> multimap =
 words.groupBy(StringFunctions.length());
```

With duplicates removed, only four 3-letter words remain.

key	value <list>
1	a

```
3               too,are,few,not,
4               long,here,that
5               words
```

Creating and converting collections

The following methods can be used to convert one container type to another. All of these methods are on ***RichIterable***. To create immutable and fixed-size collections, refer to Immutable collections.

toList()	Converts the collection to the default ***MutableList*** implementation (**FastList**).
toSet()	Converts the collection to the default ***MutableSet*** implementation (**UnifiedSet**).
toBag()	Converts the collection to the default ***MutableBag*** implementation (**HashBag**).
toMap(*keySelector*, *valueSelector***)**	Converts the collection to the default ***MutableMap*** implementation (**UnifiedMap**) using the specified keySelectors and *valueSelectors*.
toSortedList()	Converts the collection to the default ***MutableList*** implementation (**FastList**) and sorts it using the natural order of the elements.
toSortedList(Comparator**)**	Converts the collection to the default ***MutableList*** implementation (**FastList**) and sorts it using the specified Comparator.

These methods always return new *mutable* copies: for example, calling **toList()** on a **FastList**, returns a new **FastList**.

To create a new collection of the same type

newEmpty()	Creates a new, empty, and mutable container of the same collection type. For example, if this instance is a **FastList**, this method will return a new empty **FastList**. If the class of this instance is immutable (see below) or fixed size (for example, a singleton List) then a mutable alternative to the class is returned.

Protecting collections

GS Collections provides special interfaces for controlling and preventing changes to containers and their elements.

- *Immutable collection*: a copy that is permanently unchangeable.
- *Unmodifiable collection*: a read-only interface wrapped over a backing collection that remains mutable.
- *Synchronized collection*: a wrapper that presents a *mostly* thread-safe view of a collection.

Immutable collections

A read-only snapshot of a collection; once created, it can never be modified.

All of the basic containers in GS Collections have interfaces for both *mutable* and *immutable* (unchangeable) forms. This departs somewhat from the JCF model, in which most containers are mutable.

An *immutable collection* is just that - once created, it can never be modified, retaining the same internal references and data throughout its lifespan. An immutable collection is equal to a corresponding mutable collection with the same contents; a ***MutableList*** and an ImmutableList can be equal.

Because its state does not change over time, an immutable collection is *always* thread-safe. Using immutable collections where feasible can serve to make your code easier to read and understand.

All of the interfaces and implementations discussed so far in this topic have been *mutable* versions of their respective types. Each of these containers has an immutable counterpart: These are the corresponding interfaces:

Mutable types	Immutable types
MutableList	***ImmutableList***
MutableSet	***ImmutableSet***
MutableBag	***ImmutableBag***
MutableMap	***ImmutableMap***
MutableMultimap	***ImmutableMultimap***

The method that returns an immutable collection for all container types is:

MutableCollection.**toImmutable()**	Returns an immutable copy of a type corresponding to the source ***MutableCollection***.

An immutable-collection interface lacks mutating methods, such as add() and remove(). Instead, immutable collections have methods that return new, immutable copies with or without specified elements:

ImmutableCollection.**newWith(***element***)**	Returns a new immutable copy of *ImmutableCollection* with element added.
ImmutableCollection.**newWithAll(**Iterable**)**	Returns a new immutable copy of *ImmutableCollection* with the elements of Iterable added.
ImmutableCollection.**newWithout(***element***)**	Returns a new immutable copy of *ImmutableCollection* with element removed.
ImmutableCollection.**newWithoutAll(**Iterable**)**	Returns a new immutable copy of *ImmutableCollection* with the elements of Iterable removed.

Note that the iteration methods of an immutable container - such as **select**, **reject**, and **collect** - also produce new, immutable collections.

Immutable Collection Factory Classes

The factory classes **Lists**, **Sets**, **Bags**, and **Maps** create immutable collections. These factories also provide methods for creating fixed-size collections, which have been superseded by immutable collections.

```
ImmutableList<Integer> immutableList = Lists.immutable.of(1, 2, 3);
ImmutableSet<Integer> immutableSet = Sets.immutable.of(1, 2, 3);
Bag<Integer> immutableBag = Bags.immutable.of(1, 2, 2, 3);
ImmutableMap<Integer, String> immutableMap =
    Maps.immutable.of(1, "one", 2, "two", 3, "three");
```

These factories highlight yet another benefit of immutable collections: they let you create efficient containers that are sized according to their contents. In cases where there are many, even millions of collections, each with a size less than 10, this is an important advantage.

Protective wrappers

Wrapper classes providing read-only or thread-safe views of a collection.

Unmodifiable Collections

In both the JCF and GS Collections, a collection may be rendered *unmodifiable*. In GS Collections, this is done by means of the **asUnmodifiable** method, which returns a read-only view of the calling collection. This means that the mutating methods of the collection (e.g., **add**, **remove**) are still present, but throw exceptions if called.

***MutableCollection*.asUnmodifiable()**	Returns a read-only view of the source collection.

Synchronized Collections

GS Collections provides a wrapper for rendering a modifiable but thread-safe view that holds a lock when a method is called and releases the lock upon completion.

***MutableCollection*.asSynchronized()**	Returns a synchronized copy of the source collection.

Chapter 3

Code blocks

Topics:

- *Common code block types*

A *code block*, in GS Collections terms, is a single-abstract-method object that is passed as a parameter to an iteration method. It is an abstraction that represents the evaluation of each element in the course of iteration. It helps us to further separate *what* is being done from *how* it's done. This topic enumerates the basic code block types - the GS Collections interfaces and classes - and the relevant methods to which they apply.

What we call a "code block" in GS Collections is roughly analogous to what is more formally and precisely termed a *closure* or *first-class function*. A closure is a function that, when passed to a method, can access and modify variables in its enclosing scope. This is a facility that the Java language does not (as of this writing) support. The closest analog to a closure in Java is the *anonymous inner class* (which allows read access to local final variables), and this is one technique among several for implementing code blocks in GS Collections.

In this inline example, the highlighted text is a nameless code block, used as *predicate* (a yes/no function used as a filter) that implements a ***Predicate*** interface. This code block is passed as the parameter of a **select** method call. A ***Predicate*** has one method of its own, **accept(),** which takes as its sole argument **each**, the current element *upon each iteration* of the enclosing **select** method.

GSC

```
MutableList<Person> texans = this.people.select(new
 Predicate<Person>() {
  public boolean accept(Person each) {
    return each.getAddress().getState().equals("TX");
  }
});
Verify.assertSize(1, texans);
```

In this case, if the value of *state* field for any element in **people** equals "TX" then the **select** method adds that element to the new list, **texans**.

About *parameter* and *argument*:

These terms are often (if inaccurately) used interchangeably to refer to method or function inputs. (The usual distinction holds that *parameter* refers to a formal definition of the input, while argument denotes the actual values.) For the limited purposes of this guide - and in particular the scope of this topic - we use *parameter* to specify the input to an iteration method - in this example, **select**. These parameters can take the form of the code block (as described in this topic), which itself is an object with methods. The input for a code block we refer to here as the *argument* - in this example, the argument is **each** (the "current element" upon each iteration).

Common code block types

Here is a summary of the most commonly-used code blocks and the GS Collections methods that use them.

	Arguments	Returns	Used By
Predicate Evaluates each element of a collection (the argument), and returns a boolean value.	One (T)	boolean	**select**, **reject**, **detect**, **anySatisfy**, **allSatisfy**, **count**
Predicate2	Two (T,P)	boolean	**selectWith**,**rejectWith**, **detectWith**, **anySatisfyWith**,**allSatisfyWith**, **countWith**
Function *(transformer):* Evaluates each element of a collection as the argument to the code block logic and returns a computed value	One (T)	Object (V)	**collect**,**flatCollect**, **groupBy**
Function2	Two (T,P)	Object (V)	**forEachEntry() injectInto() collectWith()**
Function3	Three (T,P,?)	Object (V)	**injectIntoWith**
Procedure *:* Executes on each element of a collection, returns nothing.	One (T)	void	**forEach**, **forEachKey**, **forEachValue**,
Procedure2	Two (T,P)	void	**forEachWith**, **forEachKeyValue**
Function0 *:* Executes and returns a value (like Callable); represents deferred evaluation.	Zero	Object (V)	**getIfAbsent**, **getIfAbsentPut**, **ifPresentApply**
Comparator: "Imposes a *total ordering* on some collection of objects." (JDK)	Two (T,T)	int (negative, 0, positive)	**sortThis**, **max**, **min**

Predicate

A ***Predicate*** is a single-argument code block that evaluates an element and returns a boolean value. Also known as a *discriminator* or *filter*, it is used with the filtering methods **select**, **reject**, **detect**, **anySatisfy**, **allSatisfy**, and **count**.

The **accept** method is implemented to indicate the object passed to the method meets the criteria of this ***Predicate***.

Predicate Factories	
Predicates	Supports equal, greaterThan, lessThan, in, notIn, and, or, instanceOf, null, notNull, anySatisfy, allSatisfy, etc.
Predicates2	For ***Predicate2***s that work with methods suffixed with "with."
StringPredicates	Supports empty, notEmpty, contains, isAlpha, isNumeric, isBlank, startsWith, endsWith, matches, etc.

Predicate Factories	
IntegerPredicates	Supports isEven, isOdd, isPositive, isNegative, isZero .
LongPredicates	Supports isEven, isOdd, isPositive, isNegative, isZero.

Predicates factory

The Predicates class can be used to build common Predicates (predicates) to be used with filtering patterns. **Predicates** supports equals, not equals, less than, greater than, less than or equal to, greater than or equal to, in, not in, and, or, and numerous other predicate-type operations.

Some examples with **select()**:

GSC

```
MutableList<Integer> myList =...
MutableList<Integer> selected1 = myList.select(Predicates.greaterThan(50));
```

Function

The ***Function*** code block in its most common usage takes each element of a collection as the argument to the code-block logic. It selects an attribute from the element via a "getter" - its **valueOf()** method. It then returns a computed value or, if no evaluation logic is performed, the attribute itself.

Function code blocks are used as a parameter in a variety of common GS Collections methods:

- With the **collect** method to calculate a new value for each element of a given collection, and then return a transformed collection of the same type.
- With the **groupBy** method to generate keys for each nested collection (values) of a new Multimap.
- With the **flatCollect** method, where it must return an Iterable that gets "flattened" with other iterables, into a single collection.
- With the **Predicates** factory's **attributeOperator** methods - such as **attributeLessThanOrEqualTo** - to build ***Predicate*** (boolean) objects.

Function Factories		
Functions (static class)	**getToClass()**	**getToString()**
	getPassThru()	

Other functions

IfFunction	Supports if and else using a discriminator with Function.
CaseFunction	This allows for multi-conditional or rule based selector using ***Predicate***s (use this with guidance).

Procedure

A ***Procedure*** is a code block that performs an evaluation on its single argument and returns nothing. A ***Procedure*** is most commonly used with *ForEach* -pattern methods.

Count and calculate

CountProcedure	Apply a ***Predicate*** to an object and increment a count if it returns true.

CounterProcedure	Wrap a specified block and keeps track of the number of times it is executed.
SplitDoubleSumProcedure	Create two double sums, one for items that return true for the specified discriminator and one for items that return false. A DoubleSelector must be provided.
SplitIntegerSumProcedure	Create two integer sums, one for items that return true for the specified discriminator and one for items that return false. An IntegerSelector must be provided.
SumProcedure	Summarize the elements of a collection either via a forEach() or injectInto() call. SumProcedure returns optimized primitive blocks for specialized primitive subclasses of Function which result in less garbage created for summing primitive attributes of collections.

Return a value

ResultProcedure	Store a result to be accessed after an iteration is complete; this is useful for determining a return value from a forEach() invocation, which itself has no return value.

Control execution

ChainedProcedure	Chain together blocks of code to be executed in sequence; **ChainedProcedure** can chain ***Procedure***s, ***Function***s or ***Function2s***.
CaseProcedure	Create an object form of a case statement, which instead of being based on a single switch value is based on a list of discriminator or block combinations. For the first discriminator that returns true for a given value in the case statement, the corresponding block will be executed.
IfProcedure	Evaluate the specified block only when either discriminator returns true. If the result of evaluating the ***Predicate*** is false, and the developer has specified that there is an elseProcedure, then the elseProcedure is evaluated.
IfProcedureWithInt	Apply an index that effectively filters which objects should be used.

Modify collections and maps

CollectionAddProcedure	Add elements to the specified collection when block methods are called.
CollectionRemoveProcedure	Remove element from the specified collection when block methods are called.
ConcurrentMapOfListsPutProcedure	Use a specified ***Function*** to calculate a key for an object passed to the value method. The object is put into a ***MultiReaderFastList*** contained in the specified Map at the position of the calculated key.
MapPutProcedure	Use a specified ***Function*** to calculate the key for an object and puts the object into the specified Map at the position of the calculated key.
MultimapPutProcedure	Use a specified ***Function*** to calculate the key for an object and puts the object with the key into the specified MutableMultimap.
StringBufferProcedure	Transform to string and append elements of a collection to a StringBuffer, separated by the specified separator after trans.
StringBuilderProcedure	Transform to string and append elements of a collection to a StringBuider, separated by the specified separator.

Output a collection to a system

PrintlnProcedure	Outputs an object to a PrintSteam using println.

Chapter 4

Utility GS Collections

Topics:

- *Utility iteration patterns*
- *Parallel iteration*

GS Collections includes an assortment of static utility classes, such as **Iterate** and **ListIterate**, that provide interoperability of GS Collections iteration methods with standard Java collection classes.

Class	Used with
Iterate	Iterables (inclusive Collections)
ListIterate	List
MapIterate	Maps
ArrayIterate	Arrays
StringIterate	Strings
ParallellIterate	Parallel processing

Utility iteration patterns

Pattern		Implementation

For Each	JDK	`for (int i = 0; i < list.size(); i++)`
		`{`
		`  this.doSomething(list.get(i));`
		`}`

	Utility GSC	`Iterate.forEach(collection, new Procedure() {`
		`  public void value(Object each) {`
		`  doSomething(each);`
		`  }`
		`});`

Collect	JDK	`List<Address> addresses = new ArrayList<Address>();`
		
		`for (Person person : people)`
		`{`
		`  addresses.add(person.getAddress());`
		`}`

	Utility GSC	`Iterate.collect(collection, Function.TO_STRING_SELECTOR);`

Select	JDK	`List<Integer> greaterThanFifty = new ArrayList<Integer>();`
		`for (Integer each : list)`
		`{`
		`  if (each.intValue() > 50)`
		`  {`
		`  greaterThanFifty.add(each);`
		`  }`
		`}`

	Utility GSC	`Iterate.select(collection, Predicates.greaterThan(new Integer(50)));`

Reject	JDK	`List<Integer> notGreaterThanFifty = new ArrayList<Integer>();`
		`for (Integer each : list)`
		`{`
		`  if (each <= 50)`
		`  {`
		`  notGreaterThanFifty.add(each);`
		`  }`
		`}`

	Utility GSC	`Iterate.reject(collection, Predicates.greaterThan(new Integer(50)));`

Inject Into	JDK	`List<Integer> list = Lists.mutable.of(1, 2);`
		`int result = 5;`
		`for (int i = 0; i < list.size(); i++)`
		`{`
		`  Integer v = list.get(i);`
		`  result = result + v.intValue();`
		`}`

	Utility GSC	`Iterate.injectInto(3, Lists.mutable.of(1, 2), AddFunction.INTEGER);`
Detect	JDK	`for (int i = 0; i < list.size(); i++)`

Pattern		Implementation

		{
		    Integer v = list.get(i);
		    if (v.intValue() > 50)
		    {
		        return v;
		    }
		    return null;
		}

	Utility GSC	Iterate.detect(collection, Predicates.greaterThan(new Integer(50)));

Any Satisfy	JDK	for (int i = 0; i < list.size(); i++)
		{
		    Integer v = list.get(i);
		    if (v.intValue() > 50)
		    {
		        return true;
		    }
		}
		return false;

	Utility GSC	Iterate.anySatisfy(collection, Predicates.greaterThan(new Integer(50)));

All Satisfy	JDK	for (int i = 0; i < list.size(); i++)
		{
		    Integer v = list.get(i);
		    if (v.intValue() <= 50)
		    {
		        return false;
		    }
		}
		return true;

	Utility GSC	Iterate.allSatisfy(collection, Predicates.greaterThan(new Integer(50)));

Parallel iteration

GS Collections also provides parallel iteration that allows for optimization of data-intensive algorithms. Parallel implementations of several of the serial iteration patterns are provided out of the box. Note, however that parallel algorithms are usually not the optimal solution for the problem you are facing.

Examples

Protocol		Implementation
For Each	pseudocode	for each <element> of <collection> evaluate(<element>, <with>)
	GSC	ParallelIterate.forEach(list, aProcedure);

Select	pseudocode	create <newcollection>
		 for each <element> of <collection>
		     if condition(<element>, <with>)
		         add <element> to <newcollection>

	GSC	ParallelIterate.select(list, Predicates.greaterThan(new Integer(50)));
Collect	pseudocode	create <newcollection>

Protocol		Implementation

		`for each <element> of <collection>`
		`    <result> = transform(<element>, <with>)`
		`    add <result> to <newcollection>`

	GSC	`ParallelIterate.collect(list, aFunction);`

Reject	pseudocode	`create <newcollection>`
		`for each <element> of <collection>`
		`    if not condition(<element>)`
		`        add <element> to <newcollection>`

	GSC	`ParallelIterate.reject(list, Predicates.greaterThan(new Integer(50)));`