Collections Refueled

Stuart Marks
Core Libraries
Java Platform Group, Oracle

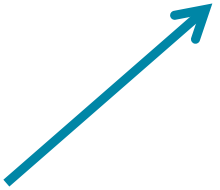
@stuartmarks

Java
Your
Next
(Cloud)

#CollectionsRefueled

Collections Refueled

- Brief History of Collections
- Java 8 Collections Enhancements
- Java 9 Collections Enhancements
- Future Collections Work



Twitter hashtag for questions and comments

(Almost) Twenty Years of Java Collections On One Slide

- JDK 1.0 – "Legacy Collections" (1996)
  - Vector, Hashtable, Enumeration, Dictionary, Stack
- JDK 1.2 – The Collections Framework (1998)
  - interfaces: Collection, List, Set, Map, Iterator, Comparable, Comparator
  - concrete classes: ArrayList, HashSet, HashMap, TreeSet, TreeMap
- Java SE 5.0 – (2004)
  - generics
  - java.util.concurrent
    - ConcurrentHashMap, CopyOnWriteArrayList, various concurrent Queues and Deques
- Other (various releases)
  - ArrayDeque, PriorityQueue, EnumMap, IdentityHashMap, LinkedHashMap, WeakHashMap

Java 8 Collections Enhancements

Java 8 Collections Enhancements

- Lambda/Streams
  - Collections are most common stream source and destination
- Interfaces: Default Methods, Static Methods
  - Java 8 language feature
  - allows interfaces to be extended compatibly
- Collection interface enhancements
  - first changes in > 15 years!
  - Iterable, Collection, List got a few new methods
  - Map, Comparator got a lot of new methods
  - Most new methods leverage lambdas and method references

Iterable Interface

- Iterable.forEach

```
// Java 7
List<String> list = ... ;
for (String str : list)
    System.out.println(str);

// Java 8
list.forEach(System.out::println);
```

- Collection is a subinterface of Iterable, so this works for all Collections

Iterable Interface

- Iterator.remove
- Most Iterators don't support removal, so everybody had to write:

```
@Override
public void remove() {
    throw new UnsupportedOperationException();
}
```

- Default implementation for remove() does exactly this
- To write a non-removing Iterator, just omit remove() !
- To write an Iterator that supports remove(), just override it as usual

Collection Interface

- Collection.removeIf – bulk mutating operation

```
// Java 7
for (Iterator<String> it = coll.iterator() ; it.hasNext() ; ) {
    String str = it.next();
    if (str.startsWith("A"))
        it.remove();
}

// Java 8
coll.removeIf(str -> str.startsWith("A"));
```

Collection Interface

- Collection.removeIf – bulk mutating operation

If collection is an ArrayList:

```
// Java 7
for (Iterator<String> it = coll.iterator() ; it.hasNext() ; ) {
    String str = it.next();
    if (str.startsWith("A"))
        it.remove();
}

// Java 8
coll.removeIf(str -> str.startsWith("A"));
```

O(n²)

O(n)

List Interface

- List.replaceAll – bulk mutation operation
- Transforms each element in-place

```
// Java 7
for (ListIterator<String> it = list.listIterator() ; it.hasNext() ; )
    it.set(it.next().toUpperCase());

// Java 7 (alt)
for (int i = 0; i < list.size(); i++)
    list.set(i, list.get(i).toUpperCase());

// Java 8
list.replaceAll(String::toUpperCase);
```

Can't change the element type.
To do that, use a stream.

List Interface

- List.sort – sorts a list in-place
- Why is this better than Collections.sort?
  - old Collections.sort used three step process:
    - copy into an temporary array
    - sort the array
    - copy back to the list
- List.sort
  - default does exactly the above
  - ArrayList.sort overrides and sorts in-place – no copying!
  - Collections.sort now just calls list.sort – callers automatically benefit

Map Interface: forEach

```
// Java 7

    for (Map.Entry<String,String> entry : map.entrySet())
        System.out.println(entry.getKey() + entry.getValue());

// Java 8

    map.forEach((k, v) -> System.out.println(k + v));
```

Map Interface: replaceAll

```
// Java 7

    for (Map.Entry<String,String> entry : map.entrySet())
        entry.setValue(entry.getValue().toUpperCase());

// Java 8

    map.replaceAll((k, v) -> v.toUpperCase());
```

Replaces map values with values of the same type. Can't change keys, key type, or value type.

Map Interface: "Multi-map" Example

- Multimap: like a map, with multiple values for each key
  - Guava and Eclipse Collections have nice implementations
- Example: simplified Multimap using Map<String, Set<Integer>>

```
Map<String, Set<Integer>> multimap = new HashMap<>();
```

  - doing this in plain Java 7 is quite painful

Map Interface: "Multi-map" Example – Java 7

```
// put(str, i)

Set<Integer> set = multimap.get(str);
if (set == null) {
    set = new HashSet<>();
    multimap.put(str, set);
}
set.add(i);

// remove(str, i)

Set<Integer> set = multimap.get(str);
if (set != null) {
    if (set.remove(i) && set.isEmpty()) {
        multimap.remove(str);
        ...                          // BLEAH!
```

Map Interface: "Multi-map" Example – Java 8

```
// put(str, i)
multimap.computeIfAbsent(str, x -> new HashSet<>()).add(i);

// remove(str, i)
multimap.computeIfPresent(k, (k1, set) -> set.remove(v) && set.isEmpty() ? null : set);

// contains(str, i)
multimap.getOrDefault(str, Collections.emptySet()).contains(i);

// size()
multimap.values().stream().mapToInt(Set::size).sum();

// values()
multimap.values().stream().flatMap(Set::stream);
```

Comparator Interface

- Anybody enjoy writing comparators?
- Comparators are difficult because there are lots of conditionals and repeated code
- Java 8 adds static and default methods to Comparator that:
  - avoid repeated code
  - allow composition of arbitrary comparators to make more complex ones
  - easily create null-friendly comparators
- Comparator example
  - two-level sort: sort by last name, then by nullable first name, nulls first

Comparator Example – Java 7

```
Collections.sort(students, new Comparator<Student>() {
    @Override
    public int compare(Student s1, Student s2) {
        int r = s1.getLastName().compareTo(s2.getLastName());
        if (r != 0)
            return r;
        String f1 = s1.getFirstName();
        String f2 = s2.getFirstName();
        if (f1 == null) {
            return f2 == null ? 0 : -1;
        } else {
            return f2 == null ? 1 : f1.compareTo(f2);
        }
    }
});
```

Comparator Example – Java 8 Statement Lambda

```
students.sort((s1, s2) -> {
    int r = s1.getLastName().compareTo(s2.getLastName());
    if (r != 0)
        return r;
    String f1 = s1.getFirstName();
    String f2 = s2.getFirstName();
    if (f1 == null) {
        return f2 == null ? 0 : -1;
    } else {
        return f2 == null ? 1 : f1.compareTo(f2);
    }
});
```

Comparator Example – Java 8 Comparator Methods

```
students.sort(comparing(Student::getLastName)
              .thenComparing(Student::getFirstName,
                             nullsFirst(naturalOrder())));
```

Comparator Example – Java 8 Comparator Methods

comparing() extracts a key and creates a Comparator that compares that key

```
students.sort(comparing(Student::getLastName)
              .thenComparing(Student::getFirstName,
                             nullsFirst(naturalOrder())));
```

thenComparing() can take two args: a key extractor, and a comparator that's used to compare the extracted keys

nullsFirst() modifies a comparator, making it null-safe, and sorting nulls before non-nulls

"natural order" is result of calling compareTo() to compare two objects of type Comparable

Java 9 Collections Enhancements

Java 9 Collections Enhancements

- History & Background
  - Java 7 Project Coin – Collection Literals proposal
  - Post Java 8 – JEP 186 Collection Literals “research JEP”
  - both were proposals to enhance the Java language
- Collections are at “arm’s length” from the Java language
  - many other languages have collections built-in
  - Java’s only built-in aggregation constructs are arrays and classes
  - higher-level collection features are delegated to libraries
  - binding language and collection libraries too tightly created design discomfort

JEP 269 – Convenience Factory Methods for Collections

- Library-only alternative to collection literals
  - no language changes
  - gets ~80% of the benefit of language changes at a tiny fraction of the cost
- Main goals
  - convenience and brevity
  - space efficiency
  - immutability
- Status
  - integrated, available in any recent JDK 9 build
  - surfaced a surprising number of API and implementation issues

JEP 269 API Overview

```
List.of()
List.of(e1)
List.of(e1, e2)          // fixed-arg overloads up to ten elements
List.of(elements...)     // varargs supports arbitrary number of elements

Set.of()
Set.of(e1)
Set.of(e1, e2)           // fixed-arg overloads up to ten elements
Set.of(elements...)      // varargs supports arbitrary number of elements

Map.of()
Map.of(k1, v1)
Map.of(k1, v1, k2, v2)   // fixed-arg overloads up to ten key-value pairs

Map.ofEntries(entry(k1, v1), entry(k2, v2), ...)     // varargs
```

static <K,V> Map<K,V>	of() Returns an immutable map containing zero mappings.
static <K,V> Map<K,V>	of(K k1, V v1) Returns an immutable map containing a single mapping.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2) Returns an immutable map containing two mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3) Returns an immutable map containing three mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4) Returns an immutable map containing four mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5) Returns an immutable map containing five mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6) Returns an immutable map containing six mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7) Returns an immutable map containing seven mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8) Returns an immutable map containing eight mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9) Returns an immutable map containing nine mappings.
static <K,V> Map<K,V>	of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10) Returns an immutable map containing ten mappings.

Examples

```
// Java 8

    List<String> stringList = Arrays.asList("a", "b", "c");
    Set<String> stringSet = new HashSet<>(Arrays.asList("a", "b", "c"));
    Map<String,Integer> stringMap = new HashMap<>();
    stringMap.put("a", 1);
    stringMap.put("b", 2);
    stringMap.put("c", 3);

// Java 9

    List<String> stringList = List.of("a", "b", "c");
    Set<String> stringSet = Set.of("a", "b", "c");
    Map<String,Integer> stringMap = Map.of("a", 1, "b", 2, "c", 3);
```

Example: Map With Arbitrary Number of Pairs

```
Map<String, TokenType> tokens = Map.ofEntries(
    entry("@",     AT),
    entry("|",     VERTICAL_BAR),
    entry("#",     HASH),
    entry("%",     PERCENT),
    entry(":",     COLON),
    entry("^",     CARET),
    entry("&",     AMPERSAND),
    entry("!",     EXCLAM),
    entry("?",     QUESTION),
    entry("$",     DOLLAR),
    entry("::",    PAAMAYIM_NEKUDOTAYIM),
    entry("=",     EQUALS),
    entry(";",     SEMICOLON)
);
```

Design and Implementation Issues

- Handling arbitrary number of mappings
- Immutability
- Iteration Order
- Nulls Disallowed
- Duplicate Handling
- Space Efficiency
- Serializability

API Design: Handling Arbitrary Number of Mappings

- List and Set have obvious varargs extensions, not so for Map
- Investigated about 15 different approaches
  - technical evaluation: “they all suck”
  - this is the case where language syntax support would be most helpful
- Criteria
  - simple, little boilerplate
  - compile-time type-safe
  - number of elements known at compile time (avoid resizing/rehashing)
  - each key and value should be adjacent in source code
  - avoid boxing

API Design: Handling Arbitrary Number of Mappings

- Solution: Map.ofEntries(Map.Entry... entries)
- Add Map.entry() static factory method returning Map.Entry
  - suitable for static import; can use

```
entry(key, value)
```

  - instead of

```
new AbstractMap.SimpleImmutableEntry<>(key, value)
```

- Satisfies all criteria except for boxing
  - maybe... the Map.Entry can be turned into a value type in the future
- Overall a reasonable compromise

Immutability

- Collections returned by the new static factory methods are immutable
- "Conventional" immutability, not "immutable persistent"
  - attempts to add, set, or remove throw UnsupportedOperationException
- Immutability is good!
  - common case: collection initialized from known values, never changed
  - automatically thread-safe
  - provides opportunities for efficiency, especially space
- No general-purpose immutable collections exist in the JDK
  - unmodifiable wrappers are a poor substitute

Randomized Iteration Order

- Iteration order for Set elements and Map keys
  - HashSet, HashMap: order is officially unspecified
  - however, usually consistent for long periods of time (> 1 JDK release cycle)
  - inadvertent order dependencies can creep into code
- Lots of code breaks when iteration order is changed
  - occasionally necessary to improve performance or fix security holes
  - lots of code probably has latent iteration order dependencies
    - bugs just waiting to happen
  - "just change this HashMap to a LinkedHashMap"
    - random bugs disappear

Randomized Iteration Order

- Solution: randomized iteration order for JEP 269 collections
  - make iteration order predictably unpredictable!
  - iteration order will be stable within a JVM instance
  - but will change from one run to the next
- Goal: “toughen up” user code to prevent iteration order dependencies
  - bugs flushed out in development and test, before production (we hope)
- Applies only to new collections implementations
  - by definition, no existing code depends on their iteration order
  - existing collections will remain the same

Nulls Disallowed

- Nulls disallowed as List or Set members, Map keys or values
  - NullPointerException thrown at creation time
- Allowing nulls in collections back in 1.2 was a mistake
  - no collection in Java 5 or later has permitted nulls
  - particularly the java.util.concurrent collections
- Why not?
  - nulls are bad! source of NPEs
  - nulls useful as sentinel values in APIs, e.g., Map.get(), Map.compute()
  - nulls useful as sentinel values for optimizing implementations

Throw Exceptions on Duplicates

- Duplicate set elements or map keys throw IllegalArgumentException
- Duplicates in a “collection literal” are most likely a programming error
- Ideally this would be detected at compile time
  - values aren’t compile-time constants
  - next best thing: fail-fast on creation at runtime
- Very few other systems do this
  - most are “last one wins”
  - Clojure and ECMAScript (strict) are notable outliers

Example: Map With Duplicate Keys

```
Map<String, TokenType> tokens = Map.ofEntries(
    entry("@",     AT),
    entry("|",     VERTICAL_BAR),
    entry("#",     HASH),
    entry("%",     PERCENT),
    entry(":",     COLON),
    entry("^",     CARET),
    entry("&",     AMPERSAND),
    entry("|",     EXCLAM),
    entry("?",     QUESTION),
    entry("$",     DOLLAR),
    entry("::",    PAAMAYIM_NEKUDOTAYIM),
    entry("=",     EQUALS),
    entry(";",     SEMICOLON)
);
```

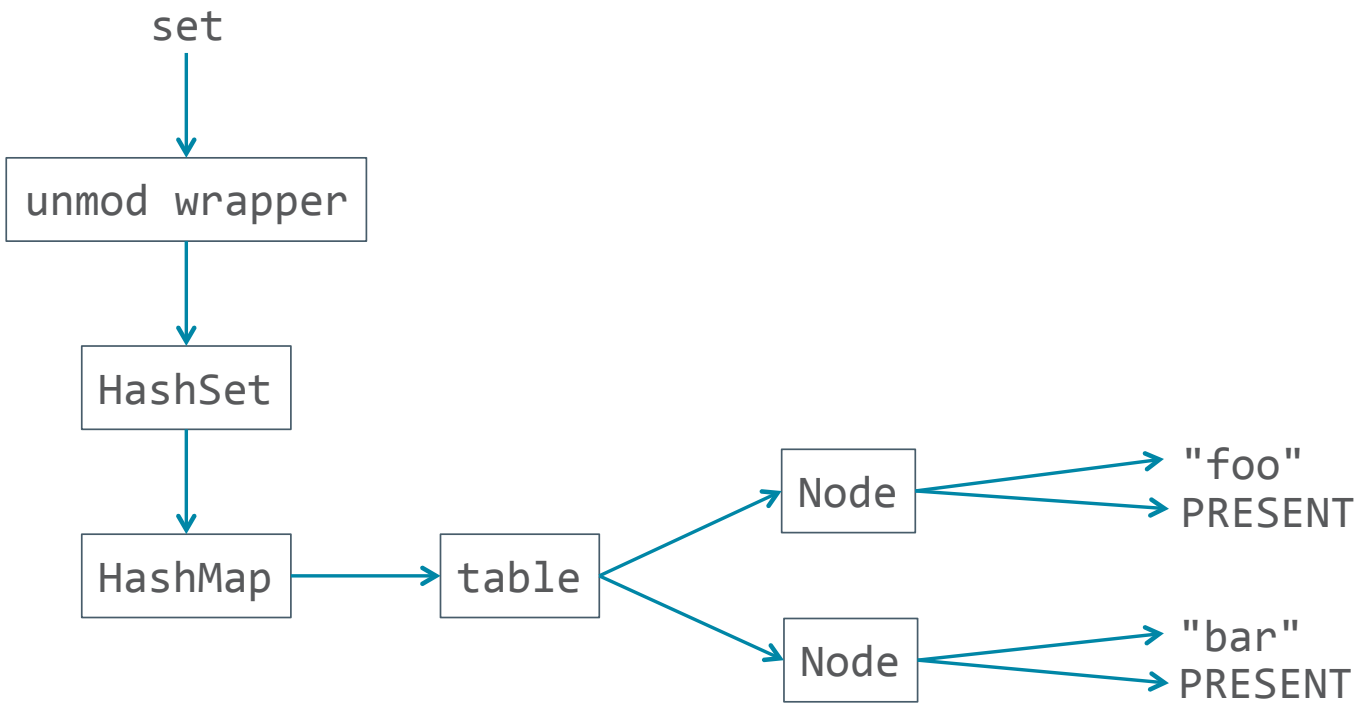
Space Efficiency

- Consider an unmodifiable set containing two strings

```
Set<String> set = new HashSet<>(3);  // 3 is the number of buckets
set.add("foo");
set.add("bar");
set = Collections.unmodifiableSet(set);
```

- How much space does this take? Count objects.
  - 1 unmodifiable wrapper
  - 1 HashSet
  - 1 HashMap
  - 1 Object[] table of length 3
  - 2 Node objects, one for each element

Space Efficiency



Space Efficiency

- Size estimate
  - 12 byte header per object
  - (assume 64-bit JVM with < 32 GB heap, allowing compressed OOPS)
  - plus 4 bytes per int, float, or reference field
- Object sizes
  - unmod wrapper: header + 1 field = 16 bytes
  - HashSet: header + 1 field = 16 bytes
  - HashMap: header + 6 fields = 36 bytes
  - table: header + 4 fields = 28 bytes
  - Node: header + 4 fields = 28 bytes x 2 = 56 bytes

Total 152 bytes to store two object references!

Space Efficiency

- Field-based set implementation

```
Set<String> set = Set.of("foo", "bar");
```

- One object, two fields
  - 20 bytes, compared to 152 bytes for conventional structure
- Efficiency gains
  - lower fixed cost: fewer objects created for a collection of any size
  - lower variable cost: fewer bytes overhead per collection element

Multiple Implementations

- All implementations are private classes hidden behind static factory
  - static factory method chooses the implementation class based on size
- Different data organizations
  - field-based implementations
    - specialized implementations for 0, 1, 2, ... elements
  - array-based with closed hashing
  - can be changed compatibly even in minor releases
- Benefits
  - less space overall
  - fewer objects result in improved locality of reference

Serialization

- All collections will be serializable
  - yes, people really use serialization
  - default serialized form would “leak” information about internal implementation
    - this can be a compatibility issue if you’re not careful
- New collections implementations will have custom serial form
  - serialization emits serial proxy to keep implementations opaque
  - deserialization chooses implementation based on current criteria in effect
  - single, common serial proxy shared by all implementations

Summary

- Collections framework is 19 years old, still useful and extensible!
- Java 8 Enhancements
  - default methods enhance all existing collections
  - Comparator methods allow building Comparators by composition
- Java 9 Enhancements – JEP 269 Convenience Factories
  - convenient, space-efficient, immutable
  - promising space savings from use in JDK 9 itself
  - JEP 269: http://openjdk.java.net/jeps/269
- Try out JDK 9 builds: http://jdk9.java.net

Safe Harbor Statement

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


ORACLE®

Future Collections Work

Short-term, Conventional Enhancements

- Deprecation
  - deprecate “legacy collections” (Vector, Hashtable, etc.)
  - deprecate LinkedList?
- Core collections enhancements
  - add opt-in randomized iteration order for core collections
  - new mutator default methods
  - indexed access for ArrayDeque (making it List-like)
- JEP 269 collections enhancements
  - improve performance
  - add ordered Set/Map

UNPLANNED FUTURE WORK

Far Distant Future – Project Valhalla

- Value types – like immutable structs, with no identity
  - "codes like a class, works like an int" – John Rose (Oracle JVM Architect)
  - includes generic specialization
- Great potential for improving conventional, mutable collections
  - collections of primitives: `List<int>`
  - uses not obvious, though; can't just replace red-black tree nodes with value types
  - possibility: reorganize internal structures to improve locality
- Immutable Persistent Collections?
  - need new APIs, can't be retrofitted into existing interfaces
  - actually orthogonal to value types

IN OTHER WORDS, HE'S LYING

Map.merge(key, newValue, (oldV, newV) -> mergeV)

- More conditional execution
- If key is absent
  - simply stores key and newValue
- If key is present
  - fetches the old value
  - invokes *merge function* on old and new values to produce merged value
  - stores the key and merged value
- Operation is atomic for ConcurrentMap implementations

Map.merge Example

```
// store or append a string to an existing value

    Map<String,String> map = new HashMap<>();

// OLD

    String oldValue = map.get("key");
    if (oldValue == null)
        map.put("key", "newValue");
    else
        map.put("key", oldValue + "newValue");

// NEW

    map.merge("key", "newValue", String::concat);
```

Collections Corner Cases – Design Notes

- Collections Framework Interfaces (Collection, List, Set, Map)
  - lasted 16 years with no modifications
  - legacy collections were all concrete classes
    - custom collections forced to use override/delegate antipattern
    - EclipseLink JPA did this with Vector to provide a laziness
    - broken in Java 8 when default methods were added
    - same things happen when people subclass/override ArrayList to customize its behavior
  - recommendation
    - always start from interfaces or one of the Abstract* classes

Collections Implementation Policies

- Policies
  - concurrent modification
    - fail-fast, snapshot, weakly consistent
    - (there is NO SUCH THING as a fail-safe iterator!)
  - iteration order
  - null handling
  - serializability
  - concurrency properties (atomicity)
- Policies are specified on implementations, not interfaces
  - if you write a custom collection, you should specify these

Optional Operations

- Some collections operations are optional
  - if not implemented, they throw UnsupportedOperationException
  - mostly for mutator methods on unmodifiable collections
- How strictly is this enforced?
  - consider: list.addAll(Collection.emptyList())
  - suppose ‘list’ is unmodifiable
  - should this throw UOE or be a no-op?
  - answer: inconsistent!
- Collections.unmodifiable* and JEP 269 factories always throw UOE
  - others, e.g. Collections.emptyList(), allow this as a no-op