Collections Refueled

Collections Framework Enhancements in Java 9

Stuart Marks
Core Libraries
Java Platform Group, Oracle

@stuartmarks

Java
Your
Next
(Cloud)

Collections Refueled

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

(Almost) Twenty Years of Java Collections

- JDK 1.0 – 1996
  - "Legacy Collections"
  - Vector, Hashtable, Enumeration, Dictionary, Stack
- JDK 1.2 – 1998
  - Collections Framework introduced
  - interfaces: Collection, List, Set, Map, Iterator, Comparable, Comparator, SortedSet, SortedMap
  - concrete classes: ArrayList, HashSet, HashMap, TreeSet, TreeMap, WeakHashMap
- JDK 1.4 – 2002
  - IdentityHashMap, LinkedHashSet, LinkedHashMap

(Almost) Twenty Years of Java Collections

- Java SE 5.0 – 2004
  - generics introduced, collections generified
  - Iterable, Queue, PriorityQueue, EnumSet, EnumMap
  - java.util.concurrent
    - ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue, etc.
- Java SE 6 – 2006
  - Deque, ArrayDeque, NavigableSet, NavigableMap (enhanced TreeSet, TreeMap)
- Java SE 7 – 2011
  - TimSort, Collections.emptyIterator ... hardly anything!

Java 8 (2014)

- Lambda/Streams
  - Collections are most common stream source and destination
- Interfaces: Default Methods, Static Methods
  - Java 8 language features
  - 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
  - Default methods ***enhanced all existing collections***

Java 8 (2014)

- Iterable.forEach
- Iterator.remove
- Iterator.forEachRemaining
- Collection.stream
- Collection.removeIf
- List.replaceAll
- List.sort
- Map.forEach
- Map.replaceAll
- Map.compute
- Map.computeIfAbsent
- Map.computeIfPresent
- Map.getOrDefault
- Comparator.comparing
- Comparator.thenComparing

Java 9 (2017)

- JEP 269 Collections Convenience Factories
  - JEP = “JDK Enhancement Proposal”
  - Static factory methods for creating collections conveniently
  - First new collection implementations since 7, first in java.util since 1.6
- Other enhancements to collections-related classes

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
- Uncovered a surprising number of API and implementation issues

JEP 269 – Convenience Factory Methods for Collections

- 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 API: Static Methods on Interfaces

```
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

Map.entry(k, v)          // creates a Map.Entry instance
```

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, contrast with Unmodifiability
- Iteration Order
- Nulls Disallowed
- Duplicate Handling
- Space Efficiency
- Serializability
- Other Behavior Differences

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

Immutability vs. Unmodifiability

- What's the difference between list1 and list2?

```
List<Integer> temp = Arrays.asList(1, 2, 3);
List<Integer> list1 = Collections.unmodifiableList(temp);
List<Integer> list2 = List.of(1, 2, 3);
```

- Similarities
  - Mutator methods add(), remove(), set() etc. throw UnsupportedOperationException
- Differences
  - list1 is an *unmodifiable view* of the underlying list temp
  - modifications to temp are visible to list1
  - list2 cannot be modified at all
    - except via reflection, but that's cheating

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 (i.e., bugs!)
  - "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
- Precedents: Go language; Python 3.0 – 3.5
- 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)
);
```

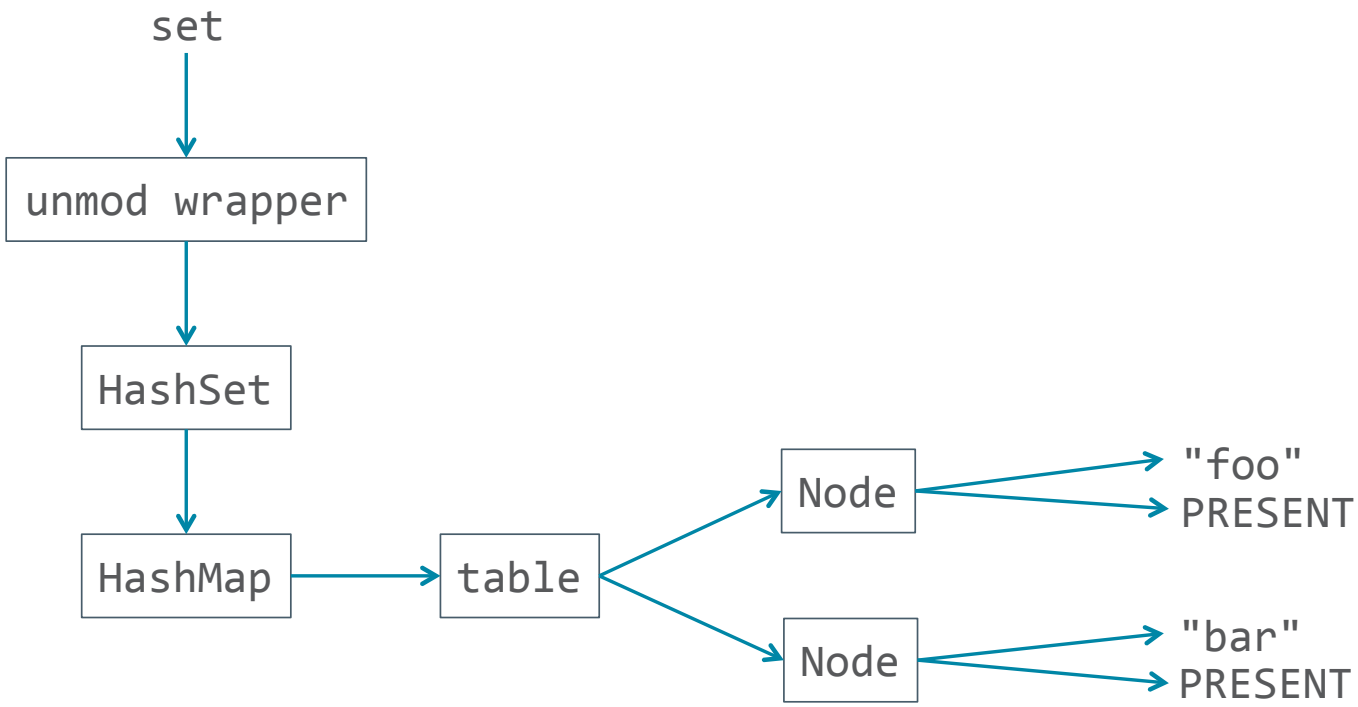
Space Efficiency

- 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 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

Other Behavior Differences

- What's the difference between list3 and list4?

```
List<Integer> list3 = Collections.singletonList(1); // immutable
List<Integer> list4 = List.of(1);
```

- Similarities
  - Mutator methods add(), remove(), set() etc. throw UnsupportedOperationException
- Differences

```
list3.addAll(Collections.emptyList());  // returns false
list4.addAll(Collections.emptyList());  // throws UOE
```

Other Vaguely Collections-Related Java 9 Enhancements

- Arrays.equals
- Arrays.compare
- Arrays.compareUnsigned
- Arrays.mismatch
- Enumeration.asIterator
- Optional.ifPresentOrElse
- Optional.or
- Optional.stream
- Scanner.tokens
- Scanner.findAll
- Matcher.replaceAll
- Matcher.replaceFirst
- Matcher.results
- Collectors.flatMapping
- Collectors.filtering
- Stream.takeWhile
- Stream.dropWhile
- Stream.ofNullable
- Stream.iterate(3-arg)

Summary

- Collections framework is 19 years old, still useful and extensible!
- Primary Java 9 Collection Enhancement: Convenience Factory Methods
  - convenient, space-efficient, immutable
  - promising space & performance improvements from use in JDK 9 itself
  - JEP 269: http://openjdk.java.net/jeps/269
- Try out JDK 9 builds: http://jdk.java.net/9/
- Me:
  - blog: stuartmarks.wordpress.com
  - Twitter: @stuartmarks

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.


Java™
ORACLE®


ORACLE®