Collections Refueled

Collections Framework Enhancements in Java 9+

Stuart W. Marks
OpenJDK Core Libraries Developer
Java Platform Group, Oracle

Twitter: @stuartmarks
Hashtag: #CollectionsRefueled

ORACLE CODE

Live for the Code

ORACLE

Safe Harbor Statement

The following 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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Twenty Years of Java Collections

- JDK 1.0 – 1996
  - "Legacy Collections": Vector, Hashtable
- JDK 1.2 – 1998
  - Collections Framework introduced: Collection, List, Set, Map, ArrayList, HashMap
- Java SE 5.0 – 2004
  - generics introduced, collections generified
  - java.util.concurrent
- Java 8 – 2014
  - lambda, streams; default methods enhanced *all existing* collections

Java 9 – Collections Convenience Factory Methods

- Convenient and Concise
- Space Efficient
- Unmodifiable

Java 9 – Collections Convenience Factory Methods

- Library-only API; no language changes
  - static factory methods for creating new lists, sets, maps
  - gets ~80% of the benefit of language changes at a tiny fraction of the cost
- Why not “collection literals” as in other languages?
  - Java’s only built-in aggregation constructs are arrays and classes
  - higher-level abstractions (collections) are delegated to libraries
  - binding Java language and libraries too tightly would create design discomfort
    - in particular, the language would now depend on collections implementations in java.util

New Java 9 APIs: 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.

List Example

```
// Java 8

    List<String> stringList =
        Collections.unmodifiableList(
            Arrays.asList("a", "b", "c"));



// Java 9

    List<String> stringList = List.of("a", "b", "c");
```

Set Example

```
// Java 8

    Set<String> stringSet =
        Collections.unmodifiableSet(
            new HashSet<>(
                Arrays.asList("a", "b", "c")));


// Java 9

    Set<String> stringSet = Set.of("a", "b", "c");
```

Map Example (<= 10 entries)

```
// Java 8

    Map<String, Integer> stringMap = new HashMap<>();
    stringMap.put("a", 1);
    stringMap.put("b", 2);
    stringMap.put("c", 3);
    stringMap = Collections.unmodifiableMap(stringMap);

// Java 9

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

Map Example (> 10 entries)

```
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)
);
```

The Map.ofEntries() method accepts a varargs argument of Map.Entry instances

Each call to entry() returns a single instance of Map.Entry

Implementation Characteristics

- Unmodifiable
- Nulls Disallowed
- Randomized Iteration Order (Sets and Maps)
- Duplicates Disallowed (Sets and Maps)
- Space Efficient
- Serializable

Unmodifiable

- Collections returned by the new static factory methods are ***unmodifiable***
  - attempts to add, set, or remove throw UnsupportedOperationException
- What good is an unmodifiable collection?
  - collections often initialized from known values, never changed
  - can pass internal collection to client without fear of accidental modification
  - one step towards thread-safety
  - provides opportunities for space efficiency
- These collections themselves are unmodifiable
  - compare Collections.unmodifiableList() etc. wrappers around another collection

Unmodifiable Collections vs. Unmodifiable Wrappers

- What's the difference between list1 and list2?

```
List<Integer> inner = Arrays.asList(1, 2, 3);
List<Integer> list1 = Collections.unmodifiableList(inner);
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 inner
  - inner can be modified, and modifications to it are visible to list1
  - list2 cannot be modified at all

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 (esp. java.util.concurrent) has permitted nulls
  - classic collections like ArrayList, HashMap still allow nulls
- Why not?
  - nulls are a source of NPEs in applications, semantically confusing
  - nulls useful as sentinel values in APIs, e.g., Map.get(), Map.compute()
  - nulls useful as sentinel values for optimizing implementations

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 new 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
- Worried? Use LinkedHashSet / LinkedHashMap

Duplicates Disallowed

- 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

- All implementations are private classes hidden behind static factory
  - static factory method chooses the implementation based on number of elements
- Different data organizations
  - field-based implementations for 0, 1, 2 elements
  - array-based with closed hashing for > 2 elements
  - implementations can be changed compatibly in any JDK release
- Benefits
  - less space overall
  - fewer objects result in improved locality of reference

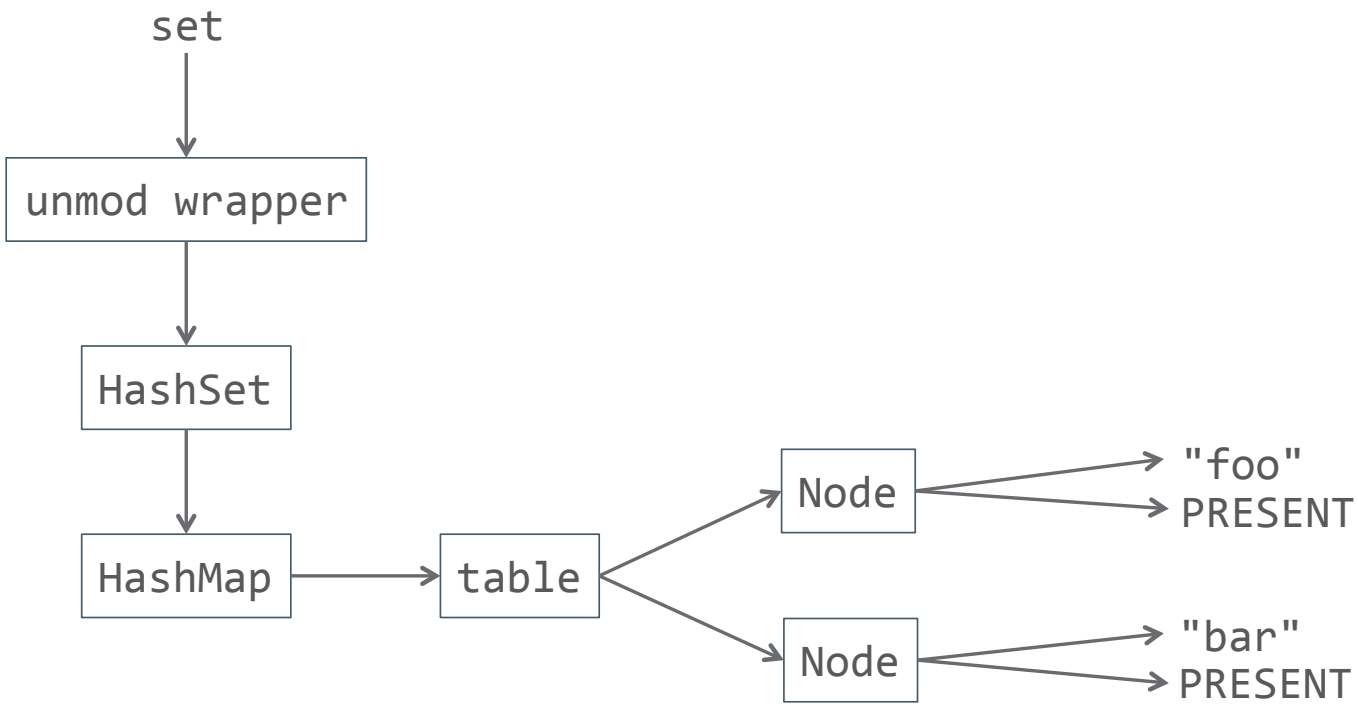
Space Efficiency

- 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

- Object size estimate
  - 12 byte header per object
  - plus 4 bytes per int, float, or reference field
  - (assume 64-bit JVM with compressed OOPS)
- Total collection overhead (not counting contents)
  - 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

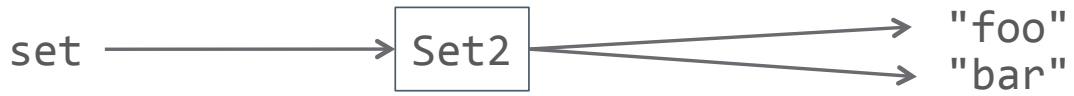
Total 152 bytes to store two object references!

Space Efficiency

- Field-based set implementation

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

- One object, two reference 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



Additional APIs in Java 10

```
// Copy Factories – for making shallow copies
// short-circuits copying if not necessary
// if src is unmodifiable, returns 'this'

List.copyOf(Collection<T> src)
Set.copyOf(Collection<T> src)
Map.copyOf(Map<K,V> src)

// Stream Collectors
// produce same implementations as List.of(), Set.of(), Map.of()

Collectors.toUnmodifiableList()
Collectors.toUnmodifiableSet()
Collectors.toUnmodifiableMap(keyFunc, valFunc)
Collectors.toUnmodifiableMap(keyFunc, valFunc, mergeFunc)
```

allows duplicates

allows duplicates

Summary

- Collections framework is 20 years old, still useful and extensible!
- Java 9 & 10 add Collection Factory Methods & Stream Collectors
  - convenient, concise API
  - space-efficient, unmodifiable implementation
  - space and performance improvements from use inthe JDK itself
- Questions?
  - Twitter: @stuartmarks #CollectionsRefueled