
ORACLE®


Java™
ORACLE

Young Pups

New Collections APIs for Java 9

Stuart Marks
Oracle Java Platform Group
Twitter: @stuartmarks

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, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Introduction

- Companion talk: “New Tricks for Old Dogs”
  - JavaOne 2015 with Mike Duigou
  - covered Java 8 enhancements to Collections
- This talk: “Young Pups”
  - new Collections enhancement proposal for Java 9
- Java 8: Lambda and Streams — Java 9: Modules
- Collections are still important!
- Tweet comments, feedback, questions on hashtag #JavaYoungPups

Java 9 Collections API Proposal & History

Collection Literals

- Other languages have collection literals
- Examples from Python:

```
stringList = ['a', 'b', 'c']
stringSet = {'a', 'b', 'c'}
stringDict = {'a':1, 'b':2, 'c':3}
```

- Why does Java have to suffer with this?

```
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);
```

We Tried: Project Coin

- Project Coin proposal, March 2009
  - Josh Bloch (the *Effective Java* guy)
  - proposed syntax for Java very similar to Python's
- Generated much interest and discussion
  - naturally, much "bikeshedding" over syntax
  - library-only alternative was discussed
- Dropped from Project Coin and Java SE 7
  - lack of time, lack of personnel
  - language changes always take longer than you think...

We Tried Again: JEP 186

- JEP 186: “Collection Literals” — a *research* JEP — Jan-Mar 2014
  - goal: decide whether to pursue language support for collection literals
  - *not* to develop the actual feature
- Two major approaches considered
  1. literals produce instances of specific, concrete types (probably private classes)
  2. literals are abstract representations of aggregations, suitable for building instances of any collection type, not just JDK collections

JEP 186 Approach #1

- Literals produce instances of specific, concrete types

  A "collection literals" feature that only worked for the built-in types (e.g., List, Set, Map), but was not extensible to user-defined types, would be disappointing.

  — Brian Goetz, lambda-dev, 2014-01-21

- Relationship between language and libraries
  - other languages (e.g., Python) are inextricably bound with collection types
  - Java the language is at "arms length" from the Java class library – loose coupling
  - fairly narrow dependencies of language on library, mostly in java.lang
  - binding the language and library too tightly causes great discomfort

JEP 186 Approach #2 and Conclusion

- Literals are abstract representations of aggregations

  The "extensible" version of this feature is open-ended, messy, and virtually guaranteed to way overrun its design budget.

  — Brian Goetz, lambda-dev, 2014-03-03

- Cost-benefit conclusion

  The library-based version gives us X% of the benefit for 1% of the cost, where X >> 1.

  — Brian Goetz, lambda-dev, 2014-03-03

(note: >> means "much greater than" not "signed right shift")

We're Trying Again: JEP 269

- Library-only alternative to collection literals
  - no language changes at this time
  - gets ~80% of the benefit of language changes
  - tiny fraction of the cost of a language change
- Current status
  - JEP 269 currently Proposed to Target for JDK 9
  - draft API under review
  - prototype in the JDK 9 "sandbox" repository

API Overview

- Goal: focus on same uses cases as collection literals
  - simplicity, brevity, convenience, but not generality
- API is indeed simple
  - uses new Java 8 feature: static methods on interfaces
  - surfaces a surprising number of API issues
  - also implementation issues

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
```

Examples

- Java (original)

```
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 (proposed)

```
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("for",      KEYWORD),
    entry("while",    KEYWORD),
    entry("do",       KEYWORD),
    entry("break",    KEYWORD),
    entry(":",        COLON),
    entry("+",        PLUS),
    entry("-",        MINUS),
    entry(">",        GREATER),
    entry("<",        LESS),
    entry("::",       PAAMAYIM_NEKUDOTAYIM),
    entry("(",        LPAREN),
    entry(")",        RPAREN)
);
```

API Design Issues

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)
  - each key and value should be adjacent
  - avoid boxing if possible

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

Overload Ambiguity: List.of(null)

- Calling List.of(null) matches
  - List.of(E) – single-element list
  - List.of(E...) – varargs
- Results in a compile-time error
- Solution: disallow nulls!
- (more discussion later)

Overload Difficulty: List.of(array)

- Suppose you have this:

```
String[] array = { "a", "b", "c" };
List.of(array);
```

- Should this create
  - a List<String> containing three strings? or
  - a List<String[]> containing a single array?
- Compiler chooses varargs, so we get List<String> with three strings
- Probably right; list of arrays is rare. Workaround:

```
List<String[]> listOfArray = List.<String[]>of(array);
```

How Many Fixed-Arg Overloads to Provide?

- No real good answer here
- Data from Google Guava:
  - frequency of use drops off steeply vs number of args
  - Guava provides 11 for list, 5 for set, 5 pairs for map
- Current proposal
  - 10 for list, 10 for set, 10 pairs for map
  - somewhat arbitrary, but intended for fixed args to catch “most” uses

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

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

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
  - fail-fast on creation at runtime
- Very few other systems do this
  - most are “last one wins”
  - Clojure and ECMAScript (strict) are notable outliers

Map Duplicate Keys

```
Map<String, TokenType> tokens = Map.ofEntries(
    entry("for",      KEYWORD),
    entry("while",    KEYWORD),
    entry("do",       KEYWORD),
    entry("break",    KEYWORD),
    entry(":",        COLON),
    entry("+",        PLUS),
    entry("-",        MINUS),
    entry(">",        GREATER),
    entry("<",        LESS),
    entry(":",        PAAMAYIM_NEKUDOTAYIM),
    entry("(",        LPAREN),
    entry(")",        RPAREN)
);
```

Implementation Design Issues

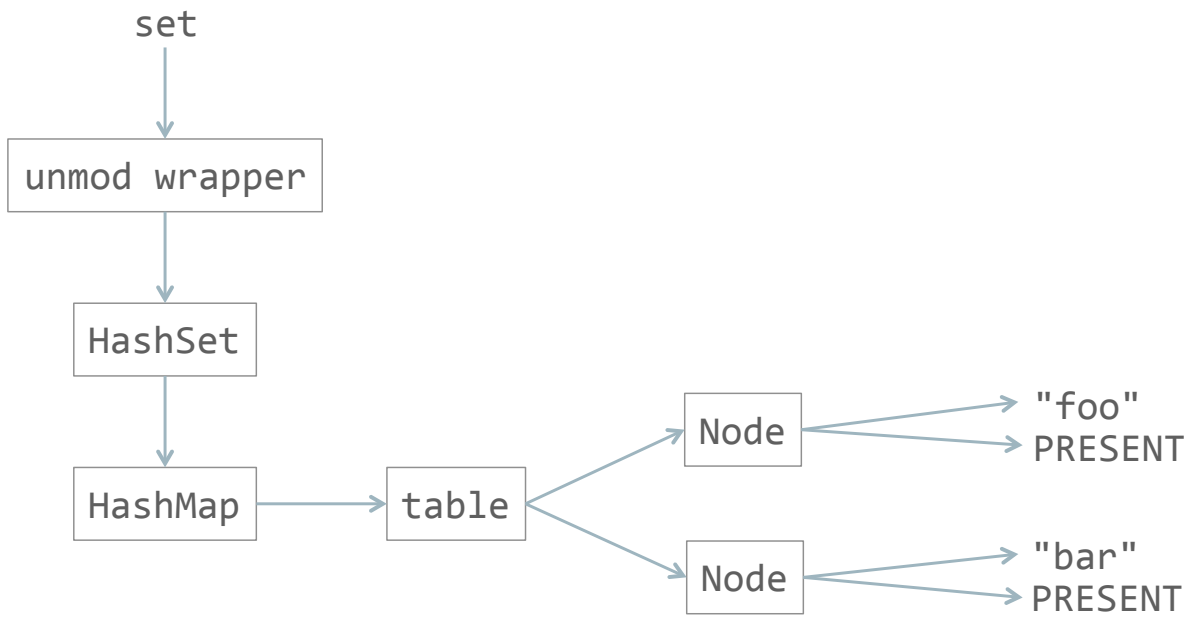
Space Efficiency

- 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: 1 field, 16 bytes
  - HashSet: 1 field, 16 bytes
  - HashMap: 6 fields, 36 bytes
  - table: 4 fields, 28 bytes
  - Node: 4 fields, 28 bytes x 2 = 56 bytes

Total 152 bytes to store two object references!

Space Efficiency

- Proposed 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

- Different data organizations
  - field-based implementations
    - specialized implementations for 0, 1, 2, ... elements
  - array-based with linear searching
  - array-based with closed hashing
  - table of bins (conventional HashMap)
- Different space vs time tradeoffs
  - array-based linear is compact but is potentially slow to search
  - array-based with hashing takes more space but is faster to search
- Fewer objects result in improved locality of reference

Multiple Implementations

- Implementation classes are private
  - no HashArraySet or LinearArrayMap in the public API
  - static factory method chooses implementation
  - primarily based on collection size
    - also possibly based on JVM object header size, padding, etc.
  - thresholds to be determined
- Implementation choices can be changed in any JDK release
  - binary compatible
  - changes possible even in minor releases

Serialization

- All collections will be serializable
  - yes, it's a pain, but people really use serialization
  - serialized form by default will "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 will emit serial proxy to keep implementations opaque
  - deserialization will choose implementation based on current criteria in effect
  - single, common serial proxy shared by all implementations

Iteration Order

- Iteration order for hashed structures has always been unspecified
- Generally, iteration order has been pretty stable
  - it's pretty common for code to have inadvertent order dependencies
  - historically, JDK has generally tried to avoid changing iteration order
  - this is often unavoidable
  - order changes once per major JDK release on average
  - when it changes, code breaks
- Different immutable collections may have different iteration orders
  - if library chooses the collection implementation
  - since implementation is opaque, iteration order might become unpredictable

Iteration Order

- Solution: randomized iteration order
  - make iteration order predictably unpredictable!
  - ideally each iteration would be different
  - practically, 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
- Applies only to new collections implementations
  - by definition, no existing code depends on their iteration order
  - existing collections will remain the same

Summary

- There's life in the old Collections Framework!
- Static factory methods proposed for JDK 9
  - convenient
  - immutable
  - space efficient
- References
  - "New Tricks for Old Dogs" (JavaOne 2015) slides available: https://stuartmarks.wordpress.com/2015/10/26/my-javaone-2015-talks-plus-recommendations/
  - JEP 269: http://openjdk.java.net/jeps/269


Java™
ORACLE


ORACLE®