ORACLE®

JavaOne™

ORACLE®

Collections:
New Tricks for Old Dogs

Stuart Marks
Oracle Java Platform Group
Twitter: @stuartmarks

Acknowledgement

Significant material in this talk was derived from the JavaOne 2014 talk of the same name, by:

Mike Duigou

OpenJDK Core Libraries Contributor

Twitter: @mjduigou

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

- Big Java 8 features were Lambda and Streams
- What about the good old Collections framework?
- Primary Java 8 Collections effort: enable collections as stream sources
  - Collection.stream() and Collection.parallelStream()
  - these are *default methods,* a new Java 8 language feature
- This talk:
  - Many other new features added to Collections in Java 8 via default methods
  - Sneak preview of features proposed for Collections in Java 9
- Tweet questions, comments, feedback with hashtag #CollectionsNewTricks

Default Methods Background

- Pre Java 8, interface methods were all abstract
  - method signature & contract (specification)
- Implementing class needed to implement all methods
- Methods basically were never added to interfaces – incompatible!
  - AbstractMethodError
- Java 8 solution to evolving an interface: default methods
  - in addition to method signature & contract, provide an *implementation*
  - inherited by all implementing classes
  - can be overridden by implementing class

Default Methods in Collections-related Interfaces

- Mostly taking advantage of Lambda expressions
- Convenience methods
- Mutating bulk operations
  - compare to streams operations, which don't mutate the source
- Transactional operations
  - multiple operations fused into a single method
  - possibly conditional
  - concurrent collections have atomic implementations
- New features automatically apply to all existing collections!

Iterable Interface

- Iterable.forEach

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

// NEW
list.forEach(s -> System.out.println(s));   // lambda
list.forEach(System.out::println);          // method reference
```

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

Iterator Interface

- Iterator.forEachRemaining
- Iterator.remove

Iterator.forEachRemaining

- Why “forEachRemaining” ?
  - can be invoked part way through an iteration
  - also, avoid name collision with Iterable.forEach
- Example: print all except first

```
Iterator<String> it = list.iterator();
if (it.hasNext())
    it.next();
it.forEachRemaining(System.out::println);
```

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() !

Collection Interface

- Collection.stream, parallelStream methods mentioned previously
- Collection.removeIf – bulk mutating operation

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

// NEW
list.removeIf(str -> str.startsWith("A"));
```

Collection.removeIf()

- Suppose the list is an ArrayList
  - (nobody uses LinkedList anymore, do they?)
- Conventional loop is O(n^2) !
  - each removal copies the tail of the array forward one position
- ArrayList.removeIf() overrides Collection.removeIf()
  - two pass algorithm
  - first pass tests each element and remembers removals in a BitSet
  - second pass removes all in one sweep
  - no element is copied more than once

List Interface

- List.replaceAll
- List.sort

List.replaceAll

- Bulk mutation operation
- Transforms each element in-place

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

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

// NEW
list.replaceAll(String::toUpperCase)
```

List.replaceAll

- Limitation: cannot change the type of the element
- If you need to change the element type, use a stream pipeline:

```
List<String> list = ... ;
List<Integer> result = list.stream()
                           .map(Integer::valueOf)
                           .collect(toList());
```

List.sort

- Sorts a List in-place
- Example

```
// OLD
Collections.sort(list, comparator);

// NEW
list.sort(comparator);
```

- Big deal! Or is it?

List.sort

- Collections.sort
  - one algorithm, must work for *all* list implementations
  - three step process
    - copy into an temporary array
    - sort the array in-place
    - copy back to the list
- List.sort
  - default does exactly the above
  - ArrayList.sort overrides and sorts in-place – no copying!
  - Collections.sort(list, cmp) now just calls list.sort(cmp) – everybody benefits!

Map Interface Enhancements

- Lots of ’em
- Simple fused operations
- Lambda-based transactional operations
- Bulk operations
- Transactional operations are atomic for ConcurrentMap implementations

Map Interface – Simple Fused Operations

- Map.getOrDefault
- Map.putIfAbsent
- Map.remove
- Map.replace(k, v)
- Map.replace(k, oldV, newV)

Map.getOrDefault(key, defaultValue)

```
// OLD

String s;
if (map.containsKey("key"))
    s = map.get("key");
else
    s = "defaultValue";


// NEW

String s = map.getOrDefault("key", "defaultValue");
```

Map.putIfAbsent(key, newValue)

```
// OLD

String s = map.get("key");
if (s == null)
    s = map.put("key", "newValue");
return s;


// NEW

String s = map.putIfAbsent("key", "newValue");
```

Map.remove(key, value)

```
// OLD

if (map.contains("key") && map.get("key").equals("value"))
    map.remove("key");


// NEW

map.remove("key", "value");
```

Map.replace(key, value)

```
// OLD

if (map.contains("key"))
    map.put("key", "value");


// NEW

map.replace("key", "value");
```

Map.replace(key, oldValue, newValue)

```
// OLD

if (map.contains("key") && map.get("key").equals("oldValue"))
    map.put("key", "newValue");


// NEW

map.replace("key", "oldValue", "newValue");
```

Map Interface – Lambda-based Operations

- New transactional operations
  - Map.compute(key, (key, oldValue) -> newValue)
  - Map.computeIfAbsent(key, key -> value)
  - Map.computeIfPresent(key, (key, oldValue) -> newValue)
  - Map.merge(key, newValue, (oldValue, newValue) -> mergedValue)
- (examples of compuetIfAbsent and merge follow)

Map.computeIfAbsent(key, key -> value)

- Conditional execution of lambda
- If key is absent
  - evaluates the lambda to get value
  - puts key & value into map
- If key is present
  - does nothing
- Operation is atomic for ConcurrentMap implementations

Map.computeIfAbsent(key, key -> value)

```
// Multi-valued map example

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

// OLD

    List<String> tempList = map.get("key");
    if (tempList == null) {
        tempList = new ArrayList<>();
        map.put("key", tempList);
    }
    tempList.add("value");

// NEW

    map.computeIfAbsent("key", k -> new ArrayList<>()).add("value");
```

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);
```

Map Interface – Bulk Operations

- Map.forEach
- Map.replaceAll

Map.forEach

```
// OLD

  for (Map.Entry<String,String> entry : map.entrySet())
      System.out.printf("key=%s value=%s%n", entry.getKey(), entry.getValue());

// NEW

  map.forEach((k, v) -> System.out.printf("key=%s value=%s%n", k, v));
```

Map.replaceAll

```
// OLD

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

// NEW

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

Comparator

- 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
- (by the way, in Java 8 interfaces can have static methods too)

Comparator Example 1

```
// Goal: sort List<Student> by last name

// OLD - anonymous inner class
    Collections.sort(students, new Comparator<Student>() {
        @Override
        public int compare(Student s1, Student s2) {
            return s1.getLastName().compareTo(s2.getLastName());
        }
    });

// NEW - use lambda expression
    students.sort((s1, s2) -> s1.getLastName().compareTo(s2.getLastName()));

// NEWER - use "comparing" utility
    students.sort(Comparator.comparing(Student::getLastName));
```

Comparator Example 2

```
// two-level sort: sort students by last name, then first name

// OLD

    students.sort((s1, s2) -> {
        int r = s1.getLastName().compareTo(s2.getLastName());
        if (r != 0)
            return r;
        return s1.getFirstName().compareTo(s2.getFirstName());
    });

// NEW

    students.sort(Comparator.comparing(Student::getLastName)
                            .thenComparing(Student::getFirstName));
```

Comparator Example 3

```
// two-level sort: sort students by last name, then by
// *nullable* first name, nulls first

// OLD

    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 3

```
// NEW

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

// NEW, static imports

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

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

Comparator Interface Enhancements Summary

- Use of *functional composition* to build complex comparators
  - instead of writing out tedious conditional logic
  - mixture of static methods and default methods
- Key extractors
  - Comparator.comparing for objects, also int, long, double
- Composition
  - Comparator.thenComparing for objects, also int, long, double
  - nullsFirst, nullsLast, reversed
- Access to natural order (for Comparable objects)
  - Comparator.naturalOrder, Comparator.reverseOrder

Java 9 Sneak Preview

- Java lacks convenient ways to create and populate collections
  - no “collection literals” like other languages
- Java lacks immutable collections
  - can use unmodifiable wrappers
  - but they aren’t really immutable
- Collections can have high per-element cost
  - also high per-collection cost
  - significant for small collections

Examples

```
// Python
letters = { 'a', 'b', 'c' }

// Java
Set<String> letters = new HashSet<>();
letters.add("a");
letters.add("b");
letters.add("c");
letters = Collections.unmodifiableSet(set);

// Java 9
Set<String> letters = Set.of("a", "b", "c");
```

#CollectionsNewTricks

Static Factory Methods Proposed for Java 9

- List
  - List.of(e1, e2, e3, ...)
- Set
  - Set.of(e1, e2, e3, ...)
- Map
  - Map.of(k1, v1, k2, v2, k3, v3, ...)
  - ok, the Map case is actually more complicated

Map Static Factory Methods

- Several fixed-arg factories up to a limit:
  - Map.of()
  - Map.of(k1, v1)
  - Map.of(k1, v1, k2, v2)
  - ...
  - Map.of(k1, v1, k2, v2, k3, v3, k4, v4, k5, v5)
- Factory method entry() for creating Map.Entry instances
  - Map factory with Map.Entry varargs parameter
  - Map.ofEntries(entry(k1, v1), entry(k2, v2), ..., entry(kN, vN))

More Examples

```
List<Integer> piDigits = List.of(3, 1, 4, 1, 5, 9, 2, 6, 5, 3);

Set<Integer> primes = Set.of(2, 7, 31, 127, 8191, 131071, 524287);


// create a map with few key-value pairs

Map<Integer, String> platonicSolids = Map.of( 4, "tetrahedron",
                                              6, "cube",
                                              8, "octahedron",
                                             12, "dodecahedron",
                                             20, "icosahedron");

// what if you have more key-value pairs than the limit?
```

#CollectionsNewTricks

Map Factory with Arbitrary Number of Pairs

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

Where are the New Collection Implementations?

- Implementations accessible *only* via the static factory methods
  - returned collection objects are all instances of private classes
- Collections from the new factories have these characteristics:
  - all are *immutable*
  - all prohibit null elements
  - set and map factories throw IllegalArgumentException on duplicates
  - sets and maps have *undefined* (and potentially *randomized*) iteration order
  - all serializable
  - space-efficient implementations
    - both per-collection and per-element

Why Immutable?

- Large set of use cases for immutability
- No need to make defensive copies
- Thread-safe by default
- Allow space efficiency optimizations
- No need for wrappers: Collection.unmodifiableList()/Set()/Map()
  - not truly immutable!
  - they are unmodifiable *views*
  - changes to underlying collection are visible

Why Prohibit Nulls?

- Allowing nulls originally was mostly considered a mistake
- Dubious semantics
  - null usually means “absent” so what does it mean if it’s present?
- None of the concurrent collections allow nulls
- No recent collections have supported nulls
- Disallowing nulls provides opportunities for optimization
  - fewer special cases in code

Why Throw Exceptions on Duplicates?

- Duplicate checking
  - elements passed to Set.of()
  - keys passed to Map.of() and Map.ofEntries()
- Factory methods are modeled on collection literals
- If you're explicitly listing all the keys or elements, duplicates are a programming error
  - catch programming errors early
  - can't check at compile time, but fail-fast at runtime

Map Factory with Arbitrary Number of Pairs

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

Spot the error...

Why Keep Implementations Private?

- Different implementations chosen based on collection size
  - e.g., field-based, linear array-based, hashed array-based
- Can change implementations from release to release
  - better algorithms
  - better tuning to current JVM and hardware characteristics
  - improvements transparent to applications
- Reduced “API footprint” means fewer compatibility worries

Summary

- Java 8 not just about Lambda and Streams!
  - many enhancements to the Collections Framework
  - go to javadoc page for your favorite collections interface
  - look under the ***Default Methods*** tab
- More to come in Java 9
  - immutable collections
  - convenient
  - null-safe
  - thread-safe
  - space efficient

JavaOne™

ORACLE®

ORACLE®