JavaOne℠

The Collections Connection
Gala Tenth (!) Edition

Joshua Bloch
Martin Buchholz
Kevin Bourrillion
Google Inc.

Goal of Talk

Find true happiness

Achieve world domination with the Java™ Collections Framework

Outline

I. What’s coming in Java 7?

II. Google Collections

III. Q & A

Pearls of wisdom from the assembled multitudes

I. What’s coming in Java 7?

- Strangely, we don’t exactly know
- Hopefully: Collection literals and array-like access
- Probably: core of Doug Lea’s Fork-Join Framework
- Probably: **`Phaser`**, **`TransferQueue`** (JSR-166y)
- Hopefully: a few goodies from Google Collections
- Improved **`sort`** *implementation*
- Improved **`ConcurrentLinkedQueue`** *implementation*

Collection Literals

Submitted to Project Coin by Josh Bloch

```
List<String> list = ["a", "b", "c"];
String firstElement = list[0];
Map<Integer, String> map = {1 : "One"};
```

Indexing syntax for Lists and Maps

Submitted to Project Coin by Shams Mahmood Imam

```
List<String> list = ...;
String firstElement = list[0];
Map<Integer, String> map = ...;
map[1] = "One";
```

Fork-Join Framework (Doug Lea)

- Framework for a style of fine-grained parallel programming in which problems are solved by (recursively) splitting them into subtasks that are solved in parallel, waiting for them to complete, and then composing results.
- A talk unto itself
  - (Which Doug gave, but not at JavaOne)

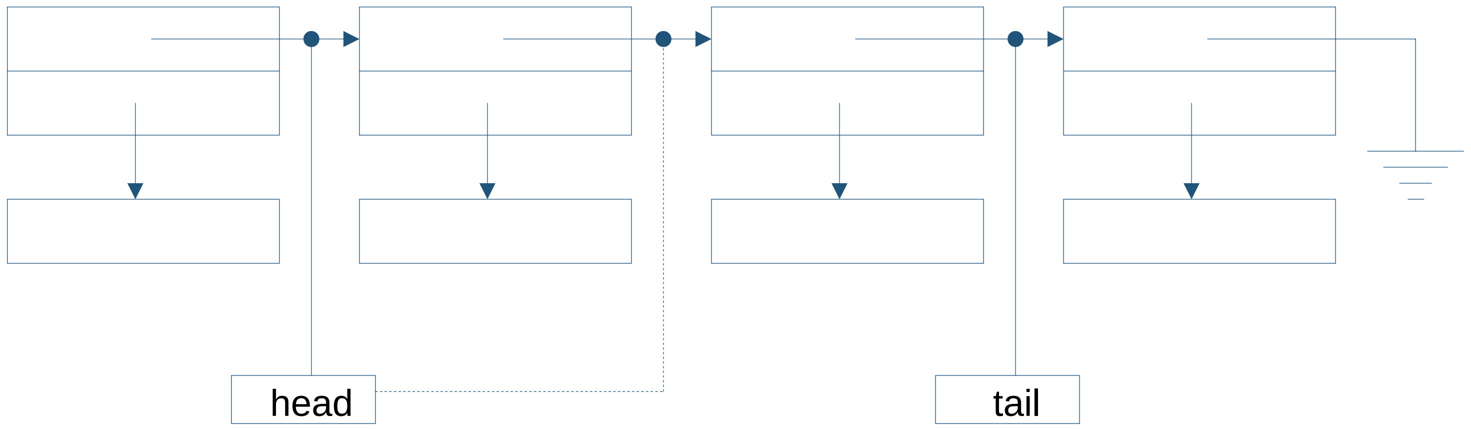
`TransferQueue`

- When **`ThreadPoolExecutor`** is below core size, it creates a new thread when request comes in, even if there is a waiting worker thread
- What is needed (internally): **`tryTransfer(Task)`**

Collections/Arrays.sort Will Use TimSort

- Adapted from Tim Peter's Python list sort
- A stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when running on partially sorted arrays, while offering performance comparable to a traditional mergesort when run on random arrays **(i.e., magic!)**
- Systematically exploits any order
- Space performance also much better than current

ConcurrentLinkedQueue Improvements



II. Google Collections

- An open-source (Apache 2) library
- Complements Java Collections Framework
  - Parts of GC will eventually migrate into JCF
  - Requires JDK 1.5
- Status: 1.0-rc2 (release candidate 2) **today**
- Widely used in production at Google
- Developers: Kevin Bourrillion & a cast of thousands

Google Collections Overview

- 1. Immutable Collections
- 2. Multisets, Multimaps
- 3. **`MapMaker`**
- Other cool interfaces, inplementations, algorithms

1. Immutable Collections

- JDK has **`Collections.unmodifiableFoo`** wrappers
- *Unmodifiable* - you can't change it
- *Immutable* - it can never change, no matter what
- Immutability is tasty!
  - See *Effective Java* Item 15 for some of the many reasons

Immutable Collections (2)

- We provide
  - **`ImmutableList`**
  - **`ImmutableSet`**
  - **`ImmutableSortedSet`**
  - **`ImmutableMap`**
  - **`ImmutableSortedMap`** (as of today!)
- Brand-new, standalone implementations

Immutable vs. unmodifiable

- JDK wrappers still useful for unmodifiable views of changing data. But for most purposes, use ours:
  - Immutability guaranteed!
  - Very easy to use
  - Slightly faster
  - Use less memory
    - Sometimes far less (ImmutableSet, factor of 2-3x)

Constant sets: Before, v1

```
public static final Set<Integer> LUCKY_NUMBERS;
  static {
    Set<Integer> set = new LinkedHashSet<Integer>();
    set.add(4);
    set.add(8);
    set.add(15);
    set.add(16);
    set.add(23);
    set.add(42);
    LUCKY_NUMBERS = Collections.unmodifiableSet(set);
  }
```

Constant sets: Before, v2

```
public static final Set<Integer> LUCKY_NUMBERS
    = Collections.unmodifiableSet(
        new LinkedHashSet<Integer>(
            Arrays.asList(4, 8, 15, 16, 23, 42)));
```

- A little nicer.
- But uses four different classes! Something's weird.

Constant sets: After

- `public static final ImmutableSet<Integer> LUCKY_NUMBERS`
  `    = ImmutableSet.of(4, 8, 15, 16, 23, 42);`
- Now we just say exactly what we mean.
- And get performance benefits as well!
- We're using just one class (it implements **`Set`**)
- **`of()`** method name inspired by java.util.EnumSet

Constant maps: Before

```
public static final Map<String, Integer> ENGLISH_TO_INT;
  static {
    Map<String, Integer> map
      = new LinkedHashMap<String, Integer>();
    map.put("four", 4);
    map.put("eight", 8);
    map.put("fifteen", 15);
    map.put("sixteen", 16);
    map.put("twenty-three", 23);
    map.put("forty-two", 42);
  ENGLISH_TO_INT = Collections.unmodifiableMap(map);
}
```

Constant maps: After

```
public static final ImmutableMap<String, Integer>
    ENGLISH_TO_INT =
        new ImmutableMap.Builder<String, Integer>()
            .put("four", 4)
            .put("eight", 8)
            .put("fifteen", 15)
            .put("sixteen", 16)
            .put("twenty-three", 23)
            .put("forty-two", 42)
            .build();
```

2. Multisets and Multimaps

- A multiset is a set that permits multiple instances
  - Set: {spam, baked-beans, sausage}
  - Multiset: [spam x 42, baked beans x 1, sausage x 1]
- A multimap is a multi-valued map
  - Map: {a=1, a=2, b=3, c=4, c=5, c=6}
  - Multimap: {a=[1, 2], b=[3], c=[4, 5, 6]}

Historically, Multimaps and Multisets Emulated Atop Maps

```
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m =
            new TreeMap<String, Integer>();
        for (String word : args) {
            Integer freq = m.get(word);
            m.put(word, (freq == null ? 1 : freq + 1));
        }
        System.out.println(m);
    }
}
```

- A bit verbose
- Bug-prone
- Uses wrong abstraction

With Multiset

```
public class Freq {
    public static void main(String[] args) {
      Multiset<String> m =
          TreeMultiset.create(Arrays.asList(args));
      System.out.println(m);
    }
}
```

- Speaks for itself!

3. **`MapMaker`** (Bob Lee)

> A **`ConcurrentMap`** builder, providing any combination of these features:
  - Soft or weak keys
  - Soft or weak values
  - Timed expiration
  - On-demand computation of values

> Far more powerful, easy to use than **`WeakHashMap`**

> Concurrent on-demand computation is **really** hard

`Mapmaker` Example

```
ConcurrentMap<Key, Graph> graphs = new MapMaker()
      .concurrencyLevel(32)
      .softKeys()
      .weakValues()
      .expiration(30, TimeUnit.MINUTES)
      .makeComputingMap(
          new Function<Key, Graph>() {
            public Graph apply(Key key) {
              return createExpensiveGraph(key);
            }
          });
```

JavaOne

MapMaker Makes it Easy to Do Hard Stuff

```
public static Comparator<Object> arbitraryOrder() {
    return new Comparator<Object>() {
        private Map<Object, Integer> uids;

        public int compare(Object left, Object right) {
            if (left == right) return 0;
            int leftCode = System.identityHashCode(left);
            int rightCode = System.identityHashCode(right);
            if (leftCode != rightCode)
                return leftCode < rightCode ? -1 : 1;

            // We have an identityHashCode collision (rare)
            synchronized (this) {
                if (uids == null)
                    final AtomicInteger counter = new AtomicInteger(0);
                    uids = new MapMaker().weakKeys().makeComputingMap(
                        new Function<Object, Integer>() {
                            public Integer apply(Object from) {
                                return counter.getAndIncrement(); }});
            }
            return uids.get(left).compareTo(uids.get(right));
        }
    };
}
```

Useful URLs

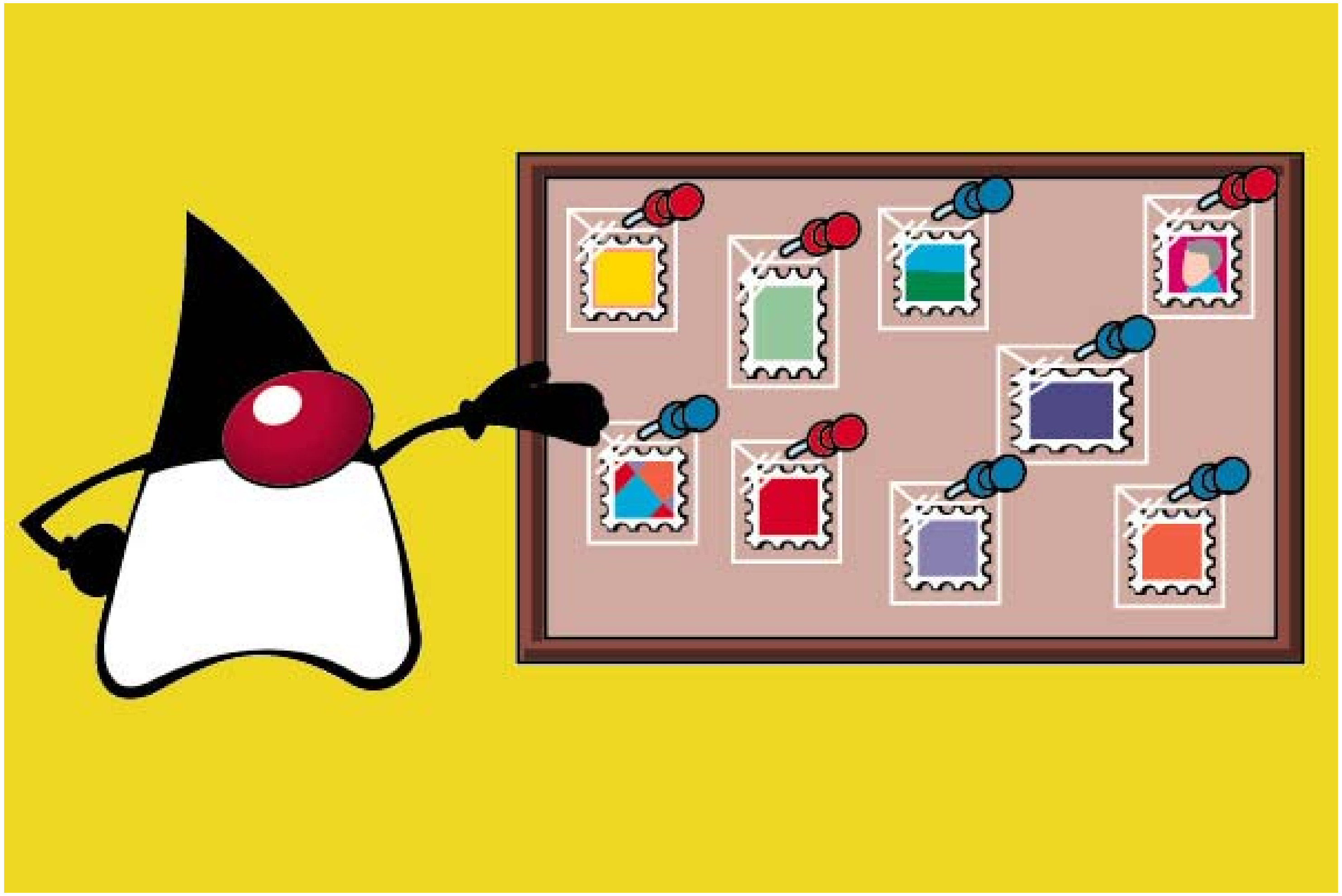
> Java Collections
  - http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html

> Google Collections
  - http://google-collections.googlecode.com

> Fork-Join Framework
  - http://g.oswego.edu/

> TimSort
  - http://svn.python.org/projects/python/trunk/Objects/listsort.txt

Obligatory Graphic



JavaOne℠

Thank You

jjb@google.com
martinrb@google.com
kevinb@google.com