Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Design case studies

Collections

Charlie Garrod       Michael Hilton
Administriva

- Homework 4b due Thursday, October 19\textsuperscript{th}  
  - Don’t wait to start!!
- Optional reading due today: Effective Java, Item 1
Early Informal Feedback

• Internal course feedback
  – Provide valuable feedback!
• 5 minutes
Review Key Concepts from Tuesday

• Design case study: GUI potpourri
  – Strategy
  – Template method
  – Observer
  – Composite
  – Decorator
  – Adapter
  – Façade
  – Command
  – Chain of responsibility
Today: Java Collections
Learning goals for today

• Understand the design aspects of collection libraries.
• Recognize the design patterns used and how those design patterns achieve design goals.
  – Marker Interface
  – Decorator
  – Factory Method
  – Iterator
  – Strategy
  – Template Method
  – Adapter
• Be able to use common collection classes, including helpers in the Collections class.
Designing a data structure library

• Different data types: lists, sets, maps, stacks, queues, ...
• Different representations
  – Array-based lists vs. linked lists
  – Hash-based sets vs. tree-based sets
  – ...
• Many alternative designs
  – Mutable vs. immutable
  – Sorted vs. unsorted
  – Accepts null or not
  – Accepts duplicates or not
  – Concurrency/thread-safe or not
  – ...

15-214
We take you back now to the late ‘90s

• It was a simpler time
  – Java had only Vector, Hashtable & Enumeration
  – But it needed more; platform was growing!
• The barbarians were pounding the gates
  – JGL was a transliteration of STL to Java
  – It had 130 (!) classes and interfaces
  – The JGL designers wanted badly to put it in the JDK
The philosophy of the Collections framework

• Powerful and general
• Small in size and conceptual weight
  – Only include fundamental operations
  – "Fun and easy to learn and use"
Java HashSet vs STL hash_set

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>Returns an iterator that addresses the first element in the hash_set.</td>
</tr>
<tr>
<td>hash_set::begin</td>
<td>Returns a const iterator addressing the first element in the hash_set.</td>
</tr>
<tr>
<td>hash_set::end</td>
<td>Returns a const iterator that addresses the location succeeding the last element in a hash_set.</td>
</tr>
<tr>
<td>clear</td>
<td>Erases all the elements of a hash_set.</td>
</tr>
<tr>
<td>count</td>
<td>Returns the number of elements in a hash_set whose key matches a parameter-specified key.</td>
</tr>
<tr>
<td>hash_set::crbegin</td>
<td>Returns a const iterator addressing the first element in a reversed hash_set.</td>
</tr>
<tr>
<td>hash_set::crend</td>
<td>Returns a const iterator that addresses the location succeeding the last element in a reversed hash_set.</td>
</tr>
<tr>
<td>hash_set::emplace</td>
<td>Inserts an element constructed in place into a hash_set.</td>
</tr>
<tr>
<td>hash_set::emplace_hint</td>
<td>Inserts an element constructed in place into a hash_set, with a placement hint.</td>
</tr>
<tr>
<td>empty</td>
<td>Tests if a hash_set is empty.</td>
</tr>
<tr>
<td>end</td>
<td>Returns an iterator that addresses the location succeeding the last element in a hash_set.</td>
</tr>
<tr>
<td>equal_range</td>
<td>Returns a pair of iterators respectively to the first element in a hash_set with a key that is greater than a specified key and to the first element in the hash_set with a key that is equal to or greater than the key.</td>
</tr>
<tr>
<td>erase</td>
<td>Removes an element or a range of elements in a hash_set by value and removes elements that match a specified key.</td>
</tr>
<tr>
<td>find</td>
<td>Returns an iterator addressing the location of an element in a hash_set that has a key equivalent to a specified key.</td>
</tr>
<tr>
<td>get_allocator</td>
<td>Returns a copy of the allocator object used to construct the hash_set.</td>
</tr>
<tr>
<td>insert</td>
<td>Inserts an element or a range of elements into a hash_set.</td>
</tr>
<tr>
<td>key_comp</td>
<td>Retrieves a copy of the comparison object used to order keys in a hash_set.</td>
</tr>
<tr>
<td>lower_bound</td>
<td>Returns an iterator to the first element in a hash_set with a key that is equal to or greater than a specified key.</td>
</tr>
<tr>
<td>max_size</td>
<td>Returns the maximum length of the hash_set.</td>
</tr>
<tr>
<td>rbegin</td>
<td>Returns an iterator addressing the first element in a reversed hash_set.</td>
</tr>
<tr>
<td>rend</td>
<td>Returns an iterator that addresses the location succeeding the last element in a reversed hash_set.</td>
</tr>
<tr>
<td>size</td>
<td>Returns the number of elements in the hash_set.</td>
</tr>
<tr>
<td>swap</td>
<td>Exchanges the elements of two hash_set.</td>
</tr>
<tr>
<td>upper_bound</td>
<td>Returns an iterator to the first element in a hash_set that has a key that is equal to or greater than a specified key.</td>
</tr>
<tr>
<td>value_comp</td>
<td>Retrieves a copy of the hash traits object used to hash and order element key values in a hash_set.</td>
</tr>
</tbody>
</table>
The java.util.Collection<E> interface

boolean add(E e);
boolean addAll(Collection<E> c);
boolean remove(E e);
boolean removeAll(Collection<E> c);
boolean contains(E e);
boolean containsAll(Collection<E> c);
void clear();
int size();
boolean isEmpty();
Iterator<E> iterator();
Object[] toArray();
E[] toArray(E[] a);
...

...
The java.util.Map<K,V> interface

Map of keys to values; keys are unique:

- V put(K key, V value);
- V get(Object key);
- V remove(Object key);
- boolean containsKey(Object key);
- boolean containsValue(Object value);
- void putAll(Map<K,V> m);
- int size();
- boolean isEmpty();
- void clear();
- Set<K> keySet();
- Collection<V> values();
- Set<Map.Entry<K,V>> entrySet();
Java Collections design decisions

• Collection represents group of elements
  – e.g. lists, queues, sets, stacks, ...

• No inherent concept of order or uniqueness

• Mutation is optional
  – May throw UnsupportedOperationException
  – Documentation describes whether mutation is supported

• Maps are not Collections

• Common functions (sort, search, copy, ...) in a separate Collections class
The `java.util.List<E>` interface

- Defines order of a collection
  - Uniqueness unspecified
- Extends `java.util.Collection<E>`:
  - `boolean add(int index, E e);`
  - `E get(int index);`
  - `E set(int index, E e);`
  - `int indexOf(E e);`
  - `int lastIndexOf(E e);`
  - `List<E> sublist(int fromIndex, int toIndex);`
The `java.util.Set<E>` interface

- Enforces uniqueness of each element in collection
- Extends `java.util.Collection<E>`:
  
  ```java
  // adds invariant (textual specification) only
  ```

- The Marker Interface design pattern
  
  - Problem: You want to define a behavioral constraint not enforced at compile time.
  
  - Solution: Define an interface with no methods, but with additional invariants as a Javadoc comment or JML specification.
The `java.util.Queue<E>` interface

- Additional helper methods only
- Extends `java.util.Collection<E>`:

  ```java
  boolean add(E e);   // These three methods
  E       remove();   // might throw exceptions
  E       element();
  
  boolean offer(E e);
  E       poll();    // These two methods
  E       peek();    // might return null
  ```
One problem: Java arrays are not Collections

• To convert a Collection to an array
  – Use the toArray() method
    List<String> arguments = new LinkedList<String>();
    ... // puts something into the list
    String[] arr = (String[]) arguments.toArray();
    String[] brr = arguments.toArray(new String[0]);

• To view an array as a Collection
  – Use the java.util.Arrays.asList() method
    String[] arr = {"foo", "bar", "baz", "qux"};
    List<String> arguments = Arrays.asList(arr);

What design pattern is this?
One problem: Java arrays are not Collections

• To convert a Collection to an array
  – Use the toArray() method
    List<String> arguments = new LinkedList<String>();
    … // puts something into the list
    String[] arr = (String[]) arguments.toArray();
    String[] brr = arguments.toArray(new String[0]);

• To view an array as a Collection
  – Use the java.util.Arrays.asList() method
    String[] arr = {"foo", "bar", "baz", "qux"};
    List<String> arguments = Arrays.asList(arr);

• The Adapter design pattern
  – Arrays.asList() returns an adapter from an array to the List interface
The abstract java.util.AbstractList<T>

abstract T get(int i); // Template Method pattern
abstract int size(); // Template Method pattern
boolean set(int i, E e); // set add remove are
boolean add(E e); // pseudo-abstract,
boolean remove(E e); // Template Methods pattern
boolean addAll(Collection<E> c);
boolean removeAll(Collection<E> c);
boolean retainAll(Collection<E> c);
boolean contains(E e);
boolean containsAll(Collection<E> c);
void clear();
boolean isEmpty();
Iterator<E> iterator();
Object[] toArray();
E[] toArray(E[] a);
...
Traversing a Collection

• Old-school Java for loop for ordered types
  ```java
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); ++i) {
      System.out.println(arguments.get(i));
  }
  ```

• Modern standard Java for-each loop
  ```java
  List<String> arguments = ...;
  for (String s : arguments) {
      System.out.println(s);
  }
  ```
  – Works for every implementation of Iterable
    ```java
    public interface Iterable<E> { 
        public Iterator<E> iterator();
    }
    ```
The Iterator interface

```java
public interface java.util.Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // removes previous returned item
} // from the underlying collection

• To use, e.g.:
  List<String> arguments = ...;
  for (Iterator<String> it = arguments.iterator();
       it.hasNext(); ) {
      String s = it.next();
      System.out.println(s);
  }
```
The Iterator design pattern

- Provide a strategy to uniformly access all elements of a container in a sequence
  - Independent of the container implementation
  - Ordering is unspecified, but every element visited once

- Design for change, information hiding
  - Hides internal implementation of container behind uniform explicit interface

- Design for reuse, division of labor
  - Hides complex data structure behind simple interface
  - Facilitates communication between parts of the program
Getting an Iterator

```java
public interface Collection<E> {
    boolean add(E e);
    boolean addAll(Collection<E> c);
    boolean remove(E e);
    boolean removeAll(Collection<E> c);
    boolean retainAll(Collection<E> c);
    boolean contains(E e);
    boolean containsAll(Collection<E> c);
    void clear();
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    Object[] toArray();
    E[] toArray(E[] a);
    ...
}
```

Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.
The Factory Method design pattern
The Factory Method design pattern

**Applicability**
- A class can’t anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates

**Consequences**
- Provides hooks for subclasses to customize creation behavior
- Connects parallel class hierarchies
Factory Method design pattern

• **Advantages:**
  – Have names
  – Not required to create a new object
  – Return an object of any subtype
  – Can reduce verbosity

• **Disadvantages**
  – Classes without public or private constructors cannot be subclassed
  – Not readily distinguishable from other static methods
Factory Method Design Pattern Usage

Collections:
   List<T>   emptyList();
   Map<K,V>  emptyMap();
   Set<T>    emptySet();

public List<String> get Countries { 
    if (/some condition/) { 
        ...
        return listCountries;
    } else { 
        return Collections.emptyList();
    }
}
An Iterator implementation for Pairs

```java
public class Pair<E> implements Iterable<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
    public Iterator<E> iterator() {
        return new PairIterator();
    }
    private class PairIterator implements Iterator<E> {
        private boolean seen1=false, seen2=false;
        public boolean hasNext() { return !seen2; }
        public E next() {
            if (!seen1) { seen1=true; return first; }
            if (!seen2) { seen2=true; return second; }
            throw new NoSuchElementException();
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
    Pair<String> pair = new Pair<String>("foo", "bar");
    for (String s : pair) { ... }
}
```
Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable...
- ...but their Iterator implementations assume the collection does not change while the Iterator is being used
  - You will get a ConcurrentModificationException
  - One way to fix:
    ```java
    List<String> arguments = ...;
    for (Iterator<String> it = arguments.iterator();
         it.hasNext(); ) {
        String s = it.next();
        if (s.equals("Michael"))
            arguments.remove("Michael"); // runtime error
    }
    ```
Sorting a Collection

• Use the Collections.sort method:
  ```java
  public static void main(String[] args) {
    List<String> lst = Arrays.asList(args);
    Collections.sort(lst);
    for (String s : lst) {System.out.println(s);}
  }
  ```

• A hacky aside: abuse the SortedSet:
  ```java
  public static void main(String[] args) {
    SortedSet<String> set =
      new TreeSet<String>(Arrays.asList(args));
    for (String s : set) { System.out.println(s); }
  }
  ```
Sorting your own types of objects

```java
public interface Comparable<T> {
    int compareTo(T o);
}
```

- **General contracts:**
  - `a.compareTo(b)` should return:
    - `<0` if `a` is less than `b`
    - `0` if `a` and `b` are equal
    - `>0` if `a` is greater than `b`
  - Should define a total order:
    - If `a.compareTo(b) < 0` and `b.compareTo(c) < 0`, then `a.compareTo(c)` should be `< 0`
    - If `a.compareTo(b) < 0`, then `b.compareTo(a)` should be `> 0`
  - Should usually be consistent with `.equals`:
    - `a.compareTo(b) == 0` iff `a.equals(b)`
Comparable objects – an example

```java
public class Integer implements Comparable<Integer> {
    private int val;
    public Integer(int val) { this.val = val; }
    ...
    public int compareTo(Integer o) {
        if (val < o.val) return -1;
        if (val == o.val) return 0;
        return 1;
    }
}
```

Aside: Is this the Template Method pattern?
Comparable objects – another example

• Make Name comparable:

```java
public class Name {
    private final String first; // not null
    private final String last;  // not null
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first;
        this.last = last;
    }
    ...
}
```

• Hint: Strings implement Comparable<String>
Comparable objects – another example

• Make Name comparable:

```java
public class Name implements Comparable<Name> {
    private final String first;  // not null
    private final String last;   // not null
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first;
        this.last = last;
    }
    ...
    public int compareTo(Name o) {
        int lastComparison = last.compareTo(o.last);
        if (lastComparison != 0) return lastComparison;
        return first.compareTo(o.first);
    }
}
```
Alternative comparisons

public class Employee implements Comparable<Employee> {
    protected Name name;
    protected int salary;
    ...
}

• What if we want to sort Employees by name, usually, but sometimes sort by salary?
Alternative comparisons

public class Employee implements Comparable<Employee> {
    protected Name name;
    protected int salary;
    ...
}

• What if we want to sort Employees by name, usually, but sometimes sort by salary?

• Answer: There's a Strategy pattern interface for that:
  public interface Comparator<T> {
    public int compare(T o1, T o2);
    public boolean equals(Object obj);
  }
void sort(int[] list, String order) {
    
    boolean mustswap;
    if (order.equals("up")) {
        mustswap = list[i] < list[j];
    } else if (order.equals("down")) {
        mustswap = list[i] > list[j];
    }
    
    ... 
}

void sort(int[] list, Comparator cmp) {
    
    boolean mustswap;
    mustswap = cmp.compare(list[i], list[j]);
    
    ... 
}

interface Comparator {
    boolean compare(int i, int j);
}

class UpComparator implements Comparator {
    boolean compare(int i, int j) { return i<j; } 
}

class DownComparator implements Comparator {
    boolean compare(int i, int j) { return i>j; } 
}
Writing a Comparator object

```java
public class Employee implements Comparable<Employee> {
    protected Name name;
    protected int salary;
    public int compareTo(Employee o) {
        return name.compareTo(o.name);
    }
}

public class EmpSalComp implements Comparator<Employee> {
    public int compare (Employee o1, Employee o2) {
        return o1.salary - o2.salary;
    }
    public boolean equals(Object obj) {
        return obj instanceof EmpSalComp;
    }
}
```
Using a Comparator

- Order-dependent classes and methods take a Comparator as an argument

```java
public class Main {

    public static void main(String[] args) {
        SortedSet<Employee> empByName = // sorted by name
            new TreeSet<Employee>();

        SortedSet<Employee> empBySal = // sorted by salary
            new TreeSet<Employee>(new EmpSalComp());
    }
}
```
The java.util.Collections class

- Standard implementations of common algorithms
  - binarySearch, copy, fill, frequency, indexOfSubList, min, max, nCopies, replaceAll, reverse, rotate, shuffle, sort, swap, ...

```java
public class Main() {
    public static void main(String[] args) {
        List<String> lst = Arrays.asList(args);
        int x = Collections.frequency(lst, "Hilton");
        System.out.println("There are " + x + " students named Hilton");
    }
}
```
The `java.util.Collections` class

- **Helper methods in `java.util.Collections`:**
  - `static List<T> unmodifiableList(List<T> lst);`
  - `static Set<T> unmodifiableSet(Set<T> set);`
  - `static Map<K,V> unmodifiableMap(Map<K,V> map);`
  - e.g., Turn a mutable list into an immutable list
    - All mutable operations on resulting list throw an `UnsupportedOperationException`

- **Similar for synchronization:**
  - `static List<T> synchronizedList(List<T> lst);`
  - `static Set<T> synchronizedSet(Set<T> set);`
  - `static Map<K,V> synchronizedMap(Map<K,V> map);`
e.g. The UnmodifiableCollection class

```java
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {
    return new UnmodifiableCollection<>(c);
}

class UnmodifiableCollection<E>
    implements Collection<E>, Serializable {

    final Collection<E> c;

    UnmodifiableCollection(Collection<> c){this.c = c; }

    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();}
    public boolean remove(Object o) {throw new UnsupportedOperationException();}
    public boolean containsAll(Collection<?> coll) {return c.containsAll(coll);
    public boolean addAll(Collection<? extends E> coll) {throw new UnsupportedOperationException();}
    public boolean removeAll(Collection<?> coll) {throw new UnsupportedOperationException();}
    public boolean retainAll(Collection<?> coll) {throw new UnsupportedOperationException();}
    public void clear() {throw new UnsupportedOperationException();}
```
The UnmodifiableCollection class

```java
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {
    return new UnmodifiableCollection<>(c);
}

class UnmodifiableCollection<E>
    implements Collection<E>, Serializable {

    final Collection<E> c;

    UnmodifiableCollection(Collection<> c){this.c = c; }

    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();}
    public boolean remove(Object o) { throw new UnsupportedOperationException();}
    public boolean containsAll(Collection<?> coll) { return c.containsAll(coll); }
    public boolean addAll(Collection<? extends E> coll) { throw new UnsupportedOperationException();}
    public boolean removeAll(Collection<?> coll) { throw new UnsupportedOperationException();}
    public boolean retainAll(Collection<?> coll) { throw new UnsupportedOperationException();}
```

What design pattern is this?
What design pattern is this?

UnmodifiableCollection class

declares Collection by removing functionality...
DISCUSSION: DESIGNING AN API (JOSH BLOCH)
Joshua Bloch

From Wikipedia, the free encyclopedia

Joshua J. Bloch (born August 28, 1961) is a software engineer and a technology author, formerly employed at Sun Microsystems and Google. He led the design and implementation of numerous Java platform features, including the Java Collections Framework, the java.math package, and the assert mechanism.[1] He is the author of the programming guide Effective Java (2001), which won the 2001 Jolt Award,[2] and is a co-author of two other Java books, Java Puzzlers (2005) and Java Concurrency In Practice (2006).

Bloch holds a B.S. in computer science from Columbia University and a Ph.D. in computer science from Carnegie Mellon University.[1] His 1990 thesis was titled A Practical Approach to Replication of Abstract Data Objects[3] and was nominated for the ACM Distinguished Doctoral Dissertation Award.[4]

Bloch has worked as a Senior Systems Designer at Transarc, and later as a Distinguished Engineer at Sun Microsystems. In June 2004, he left Sun and became Chief Java Architect at Google.[5] On August 3, 2012, Bloch announced that he would be leaving Google.[6]

In December 2004, Java Developer's Journal included Bloch in its list of the "Top 40 Software People in the World".[7]

Bloch has proposed the extension of the Java programming language with two features: Concise Instance Creation Expressions (CICE) (coproposed with Bob Lee and Doug Lea) and Automatic Resource Management (ARM) blocks. The combination of CICE and ARM formed one of the three early proposals for adding support for closures to Java.[8] ARM blocks were added to the language in JDK7.[9]

Bloch is currently a faculty member of the Institute for Software Research at Carnegie Mellon University, where he holds the title "Professor of the Practice". In addition to his research, Bloch teaches coursework in Software Engineering, course 15-214.
Joshua J. Bloch (born August 28, 1961) is a software engineer and a technology author, formerly employed at Sun Microsystems and Google. He led the design and implementation of numerous Java platform features, including the Java Collections Framework, the java.math package, and the assert mechanism.\[^1\] He is the author of the programming guide *Effective Java* (2001), which won the 2001 Jolt Award,\[^2\] and is a co-author of two other Java books, *Java Puzzlers* (2005) and *Java Concurrency In Practice* (2006).

In December 2004, *Java Developer's Journal* included Bloch in its list of the "Top 40 Software People in the World".\[^7\]

Bloch has proposed the extension of the Java programming language with two features: Concise Instance Creation Expressions (CICE) (coproposed with Bob Lee and Doug Lea) and Automatic Resource Management (ARM) blocks. The combination of CICE and ARM formed one of the three early proposals for adding support for closures to Java.\[^8\] ARM blocks were added to the language in JDK7.\[^9\]

Bloch is currently a faculty member of the Institute for Software Research at Carnegie Mellon University, where he holds the title "Professor of the Practice". In addition to his research, Bloch teaches coursework in Software Engineering, course 15-214.
A wonderful source of use cases

“Good artists copy, great artists steal.” — Pablo Picasso
You must maintain an *issues list*

- Centralizes all open and closed design issues
- List pros and cons for each possible decision
- Essential for efficient progress
- Forms the basis of a design rationale
The first draft of API was not so nice

- Map was called Table
- No HashMap, only Hashtable
- No algorithms (Collections, Arrays)
- Contained some unbelievable garbage
Automatic alias detection
A horrible idea that died on the vine

/**
 * This interface must be implemented by Collections and Tables that are
 * views on some backing collection. (It is necessary to
 * implement this interface only if the backing collection is not
 * encapsulated) by this Collection or Table; that is, if the
 * backing collection might conceivably be accessed in some way other
 * than through this Collection or Table.) This allows users
 * to detect potential aliasing between collections.
 * If a user attempts to modify one collection
 * while iterating over another, and they are in fact views on
 * the same backing object, the iteration may behave erratically.
 * However, these problems can be prevented by recognizing the
 * situation, and "defensively copying" the Collection over which
 * iteration is to take place, prior to the iteration.
 */

public interface Alias {
    /**
     * Returns the identityHashCode of the object "ultimately backing" this
     * collection, or zero if the backing object is undefined or unknown.
     * The purpose of this method is to allow the programmer to determine
     * when the possibility of aliasing exists between two collections
     * (in other words, modifying one collection could affect the other). This
     * is critical if the programmer wants to iterate over one collection and
     * modify another; if the two collections are aliases, the effects of
     * the iteration are undefined, and it could loop forever. To avoid
     * this behavior, the careful programmer must "defensively copy" the
     * collection prior to iterating over it whenever the possibility of
     * aliasing exists.
     * If this collection is a view on an Object that does not implement
     * Alias, this method must return the backingObjectId of the backing
     * Object. For example, a List backed by a user-provided array would
     * return the IdentityHashCode of the array.
     */
    int backingObjectId();

    // Automatic	alias	detection
    // A	horrible	idea	that	died	on	the	vine

    // If this collection is a view on another Object that implements
    // Alias, this method must return the backingObjectId of the backing
    // Object. (To avoid the cost of recursive calls to this method, the
    // backingObjectId may be cached at creation time).
    // For all collections backed by a particular "external data source" (a
    // SQL database, for example), this method must return the same value.
    // The IdentityHashCode of a "proxy" Object created just for this
    // purpose will do nicely, as will a pseudo-random integer permanently
    // associated with the external data source.
    // For any collection backed by multiple Objects (a "concatenation
    // view" of two lists, for instance), this method must return zero.
    // Similarly, for any view collection for which it cannot be
    // determined what Object backs the collection, this method must return
    // zero. It is always safe for a collection to return zero as its
    // backingObjectId, but doing so when it is not necessary will lead to
    // inefficiency.
    // The possibility of aliasing between two collections exists iff
    // any of the following conditions are true:
    // - The two collections are the same Object.
    // - Either collection implements Alias and has a
      backingObjectId that is the IdentityHashCode of
      the other collection.
    // - Either collection implements Alias and has a
      backingObjectId of zero.
    // - Both collections implement Alias and they have equal
      backingObjectId's.
    // @see java.lang.System#identityHashCode
    // @since JDK1.2
*/

15-214
I received a *lot* of feedback

- Initially from a small circle of colleagues
  - Some *very* good advice
  - Some not so good
- Then from the public at large: beta releases
  - Hundreds of messages
  - Many API flaws were fixed in this stage
  - I put up with a lot of flaming
### Review from a **very** senior engineer

<table>
<thead>
<tr>
<th>API</th>
<th>vote</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>yes</td>
<td>But remove binarySearch* and toList</td>
</tr>
<tr>
<td>BasicCollection</td>
<td>no</td>
<td>I don't expect lots of collection classes</td>
</tr>
<tr>
<td>BasicList</td>
<td>no</td>
<td>see List below</td>
</tr>
<tr>
<td>Collection</td>
<td>yes</td>
<td>But cut toArray</td>
</tr>
<tr>
<td>Comparator</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>DoublyLinkedList</td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td>HashSet</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>LinkedList</td>
<td>no</td>
<td>(without generics this isn't worth it)</td>
</tr>
<tr>
<td>List</td>
<td>no</td>
<td>I'd like to say yes, but it's just way bigger than I was expecting</td>
</tr>
<tr>
<td>RemovalEnumeration</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>yes</td>
<td>BUT IT NEEDS A DIFFERENT NAME</td>
</tr>
<tr>
<td>TreeSet</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

I'm generally not keen on the toArray methods because they add complexity.

Similarly, I don't think that the table Entry subclass or the various views mechanisms carry their weight.
III. Evolution of Java collections

<table>
<thead>
<tr>
<th>Release, Year</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 1.0, 1996</td>
<td>Java Released: Vector, Hashtable, Enumeration</td>
</tr>
<tr>
<td>JDK 1.1, 1996</td>
<td>(No API changes)</td>
</tr>
<tr>
<td>J2SE 1.2, 1998</td>
<td>Collections framework added</td>
</tr>
<tr>
<td>J2SE 1.3, 2000</td>
<td>(No API changes)</td>
</tr>
<tr>
<td>J2SE 1.4, 2002</td>
<td>LinkedHashMap{Map,Set}, IdentityHashSet, 6 new algorithms</td>
</tr>
<tr>
<td>J2SE 5.0, 2004</td>
<td>Generics, for-each, enums: generified everything, Iterable Queue, Enum{Set,Map}, concurrent collections</td>
</tr>
<tr>
<td>Java 6, 2006</td>
<td>Deque, Navigable{Set,Map}, newSetFromMap, asLifoQueue</td>
</tr>
<tr>
<td>Java 7, 2011</td>
<td>No API changes. Improved sorts &amp; defensive hashing</td>
</tr>
<tr>
<td>Java 8, 2014</td>
<td>Lambdas (+ streams and internal iterators)</td>
</tr>
</tbody>
</table>
V. Critique

Some things I wish I’d done differently

• Algorithms should return collection, not void or boolean
  – Turns ugly multiliners into nice one-liners
  ```java
  private static String alphabetize(String s) {
    return new String(Arrays.sort(s.toCharArray()));
  }
  ```
  • Collection should have get(), remove()
    – Queue and Deque eventually did this

• Sorted{Set,Map} should have proper navigation
  – Navigable{Set,Map} are warts
Conclusion – Josh Bloch

• **It takes a lot of work to make something that appears obvious**
  – Coherent, unified vision
  – Willingness to listen to others
  – Flexibility to accept change
  – Tenacity to resist change
  – Good documentation!

• **It’s worth the effort!**
  – A solid foundation can last two+ decades