Principles of Software Construction:

How to Design a Good API & Why it Matters Pt. 2
The Design of the Collections API – Pt. 1

Josh Bloch Charlie Garrod

School of
Computer Science

institute for
SOFTWARE
RESEARCH

Administrivia

- Homework 4b due **Thursday**
- HW 4a feedback (still) available after class

institute for SOFTWARE RESEARCH

Key concepts from Thursday…

- API design is critical; we are all API designers
- A process for API design
  - Gather requirements skeptically
  - Start small, circulate widely, revise repeatedly
- There are general principles for good API design
  - We discussed them as they apply to Classes
  - We will now move on to Methods and Exceptions

Review – characteristics of a good API

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

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
- It fell to me to design something better ☺

isr institute for SOFTWARE RESEARCH

Here's the first collections talk ever

- Debuted at JavaOne 1998
- No one knew what a collections framework was
  - Or why they needed one
- Talk aimed to
  - Explain the concept
  - Sell Java programmers on this framework
  - Teach them to use it

The Java™ Platform Collections Framework

Joshua Bloch

Sr. Staff Engineer, Collections Architect

Sun Microsystems, Inc.

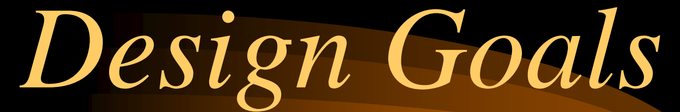
- Object that groups elements
- Main Uses
  - Data storage and retrieval
  - Data transmission
- Familiar Examples
  - java.util.Vector
  - java.util.Hashtable
  - array

What is a Collections Framework?

- Unified Architecture
  - Interfaces - implementation-independence
  - Implementations - reusable data structures
  - Algorithms - reusable functionality
- Best-known examples
  - C++ Standard Template Library (STL)
  - Smalltalk collections

Benefits

- Reduces programming effort
- Increases program speed and quality
- Interoperability among unrelated APIs
- Reduces effort to learn new APIs
- Reduces effort to design new APIs
- Fosters software reuse



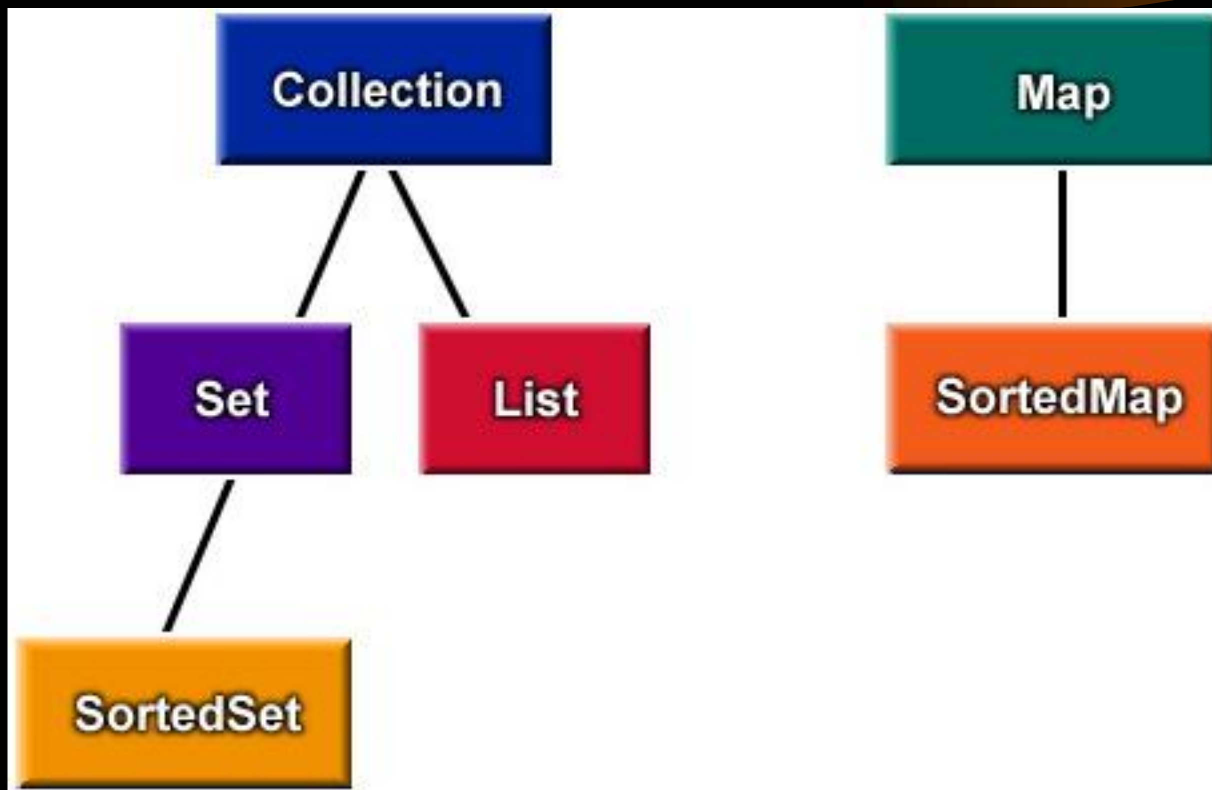
Design Goals

- Small and simple
- Reasonably powerful
- Easily extensible
- Compatible with preexisting collections
- Must feel familiar

Architecture Overview

- Core Collection Interfaces
- General-Purpose Implementations
- Wrapper Implementations
- Abstract Implementations
- Algorithms

Core Collection Interfaces



Collection Interface

```
public interface Collection {
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element);    // Optional
   boolean remove(Object
   Iterator iterator();

   Object[] toArray();
   Object[] toArray(Object a[]);

   // Bulk Operations
   boolean containsAll(Collection c);
   boolean addAll(Collection c);    // Optional
   boolean removeAll(Collection c); // Optional
   boolean retainAll(Collection c); // Optional
   void clear();                    // Optional
}
```

Iterator Interface

- Replacement for Enumeration interface
  - Adds `remove` method
  - Improves method names

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();      // Optional
}
```

Collection Example

```
public static boolean removeNulls(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext(); ) {
    if (i.next() == null)
      i.remove();
  }
}
```

Set Interface

- Adds no methods to Collection!
- Adds stipulation: no duplicate elements
- Mandates equals and hashCode calculation

```
public interface Set extends Collection {
}
```

Set Idioms

```
Set s1, s2;
boolean isSubset = s1.containsAll(s2);
Set union = new HashSet(s1);
union = union.addAll(s2); [sic]
Set intersection = new HashSet(s1);
intersection = intersection.retainAll(s2);[s]
Set difference = new HashSet(s1);
difference = difference.removeAll(s2); [sic]
Collection c;
Collection noDups = new HashSet(c);
```

List Interface

A sequence of objects

```
public interface List extends Collection {
   Object get(int index);
   Object set(int index, Object element); // Optional
   void add(int index, Object element);   // Optional
   Object remove(int index);              // Optional
   abstract boolean addAll(int index, Collection c);
                                          // Optional
   int indexOf(Object o);
   int lastIndexOf(Object o);

   List subList(int from, int to);

   ListIterator listIterator();
   ListIterator listIterator(int index);
}
```

List Example

Reusable algorithms to swap elements and randomize

```
public static void swap(List a, int i, int j) {
  Object tmp = a.get(i);
  a.set(i, a.get(j));
  a.set(j, tmp);
}

private static Random r = new Random();

public static void shuffle(List a) {
  for (int i=a.size(); i>1; i--)
    swap(a, i-1, r.nextInt(i));
}
```

List Idioms

```
List a, b;
// Concatenate two lists
a.addAll(b);
// Range-remove
a.subList(from, to).clear();
// Range-extract
List partView = a.subList(from, to);
List part = new ArrayList(partView);
partView.clear();
```

Map Interface

A key-value mapping

```
public interface Map {
   int size();
   boolean isEmpty();
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   Object get(Object key);
   Object put(Object key, Object value);    // Optional
   Object remove(Object key);               // Optional

   void putAll(Map t);   // Optional
   void clear();         // Optional

  // Collection Views
   public Set keySet();
   public Collection values();
   public Set entrySet();

}
```

Map Idioms

```
// Iterate over all keys in Map m
Map m;
for (iterator i = m.keySet().iterator(); i.hasNext(); )
     System.out.println(i.next());

// "Map algebra"
Map a, b;

boolean isSubMap = a.entrySet().containsAll(b.entrySet());

Set commonKeys = new
 HashSet(a.keySet()).retainAll(b.keySet);

//Remove keys from a that have mappings in b
a.keySet().removeAll(b.keySet());

// Etc.!!!
```

General Purpose Implementations

Consistent Naming and Behavior

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		Linked List
	Map	HashMap		TreeMap	

Choosing an Implementation

- Set
  - HashSet -- O(1) access, no order guarantee
  - TreeSet -- O(log n) access, sorted
- Map
  - HashMap -- (See HashSet)
  - TreeMap -- (See TreeSet)
- List
  - ArrayList -- O(1) random access, O(n) insert/remove
  - LinkedList -- O(n) random access, O(1) insert/remove;
  - Use for queues and deques

Implementation Behavior

- Fail-fast iterator
- Null elements, keys, values permitted
- Not thread-safe

Synchronization Wrappers

A New Approach to Thread Safety

- Anonymous implementations, one per core interface
- Static factories take collection of appropriate type
- Thread-safety assured if all access through wrapper
- Must manually synchronize iteration

Synchronization Wrapper Example

```
Set s = Collections.synchronizedSet(new HashSet());
        ...
s.add("wombat");  // Thread-safe
        ...
synchronized(s) {
    Iterator i = s.iterator(); // In synch block!
    while (i.hasNext())
        System.out.println(i.next());
}
```

Unmodifiable Wrappers

- Analogous to synchronization wrappers
  - Anonymous implementations
  - Static factory methods
  - One for each core interface
- Provide read-only access

Convenience Implementations

- **`Arrays.asList(Object[] a)`**
  - Allows array to be "viewed" as List
  - Bridge to Collection-based APIs
- **`EMPTY_SET, EMPTY_LIST, EMPTY_MAP`**
  - immutable constants
- **`singleton(Object o)`**
  - immutable set with specified object
- **`nCopies(Object o)`**
  - immutable list with n copies of object

Custom Implementation Ideas

- Persistent
- Highly concurrent
- High-performance, special-purpose
- Space-efficient representations
- Fancy data structures
- Convenience classes

Custom Implementation Example

It's easy with our abstract implementations

```
// List adapter for primitive int array
public static List intArrayList(final int[] a) {
  return new AbstractList() {
    public Object get(int i) {
      return new Integer(a[i]);
    }

    public int size() {return a.length;}

    public Object set(int i, Object o) {
      int oldVal = a[i];
      a[i] = ((Integer)o).intValue();
      return new Integer(oldVal);
    }
  };
}
```

Reusable Algorithms

- `static void sort(List[]);`
- `static int binarySearch(List list, Object key);`
- `static object min(List[]);`
- `static object max(List[]);`
- `static void fill(List list, Object o);`
- `static void copy(List dest, List src);`
- `static void reverse(List list);`
- `static void shuffle(List list);`

Algorithm Example 1

Sorting Lists of Comparable Elements

```
List strings;                  // Elements type: String
    ...
Collections.sort(strings); // Alphabetical order


Vector dates;                  // Elements type: Date
    ...
Collections.sort(dates);   // Chronological order


// Comparable interface (Infrastructure)
public interface Comparable {
    int compareTo(Object o);
}
```

Comparator Interface

(Infrastructure)

- Specifies order among objects
  - Override *natural order* on Comparables
  - Provide order on non-Comparables

```
public interface Comparator {
    public int compare(Object o1, Object o2);
}
```

Algorithm Example 2

Sorting with a Comparator

```
List strings; // Element type: String
   ...
Collections.sort(strings, Collections.ReverseOrder());


// Case-independent alphabetical order
static Comparator cia = new Comparator() {
  public int compare(Object o1, Object o2) {
    return ((String)o1).toLowerCase().
     compareTo(((String)o2).toLowerCase());
  }
};
Collections.sort(strings, cia);
```

Compatibility

Old and new collections interoperate freely

- Upward Compatibility
  - `Vector implements List`
  - `Hashtable implements Map`
  - `Arrays.asList(myArray)`
- Backward Compatibility
  - `myCollection.asArray()`
  - `Vector(myCollection)`
  - `Hashtable(myMap)`

API Design Guidelines

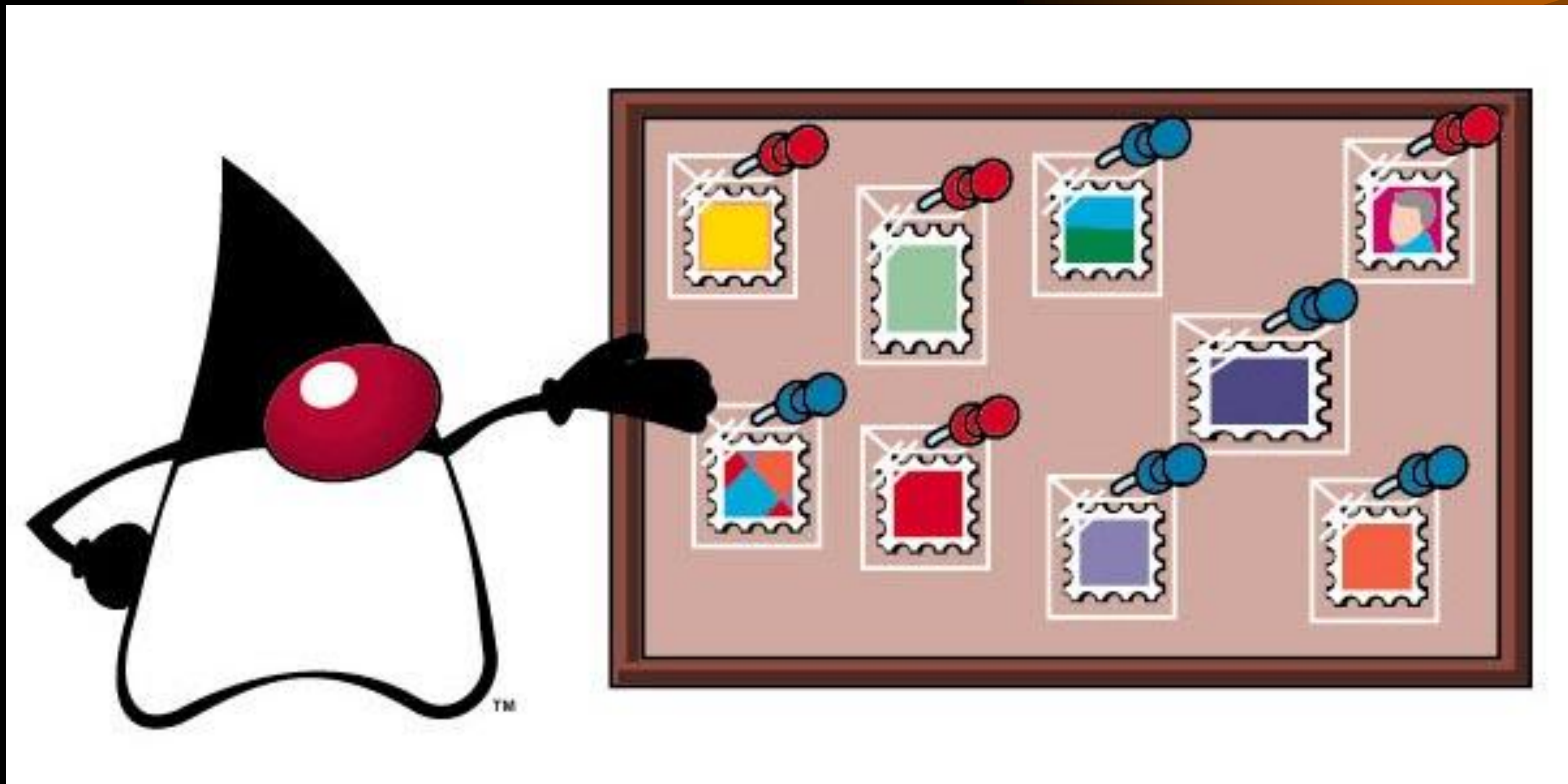
- Avoid ad hoc collections
  - Input parameter type:
    - Any collection interface (Collection, Map best)
    - Array may sometimes be preferable
  - Output value type:
    - Any collection interface or type
    - Array
- Provide adapters for legacy collections

Sermon

- Programmers:
  - Use new implementations and algorithms
  - Write reusable algorithms
  - Implement custom collections

- API Designers:
  - Take collection interface objects as input
  - Furnish collections as output

For More Information



http://java.sun.com/products/jdk/1.2/docs/guide/collections/index.html

Takeaways

- Collections haven't changed that much since '98
- API has grown, but essential character unchanged
- PowerPoint templates sure were ugly back then
- Caveat: don't use *raw types* as this talk did
  - Use `Set<String>`, `Set<E>`, or `Set<?>`; not `Set`
  - `Generics came six years after this talk`

Come back Thursday for:

- *How* the collections framework was designed
  - I found a trove of documents from 1997
  - The first draft was much uglier!
- How the collections framework evolved
  - What has been added over the past two decades
  - How it fits in to the original design
- Critique
  - What I wish I'd done differently

isr institute for SOFTWARE RESEARCH