Principles of Software Construction: Objects, Design, and Concurrency

Design case studies

The Java Collections Framework

Josh Bloch Charlie Garrod Darya Melicher

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4b due next Thursday, October 17th
  - Homework 4a feedback coming tomorrow or Thursday

Key concepts from last Thursday

- Use the observer pattern to decouple two-way dependences
- Multi-threaded programming is genuinely hard
  - Neither under- nor over-synchronize
  - Immutable types are your friend
- GUI programming is inherently multi-threaded
  - Swing calls must be made on the event dispatch thread
  - No other significant work should be done on the EDT

Key concepts from last recitation

- GUI programming is not pretty
- GUI programming is a bit tedious
- But it’s not magic, and you can do it

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 ☺

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^{TM} Platform Collections Framework

Joshua Bloch

Sr. Staff Engineer, Collections Architect

Sun Microsystems, Inc.

What is a Collection?

- 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

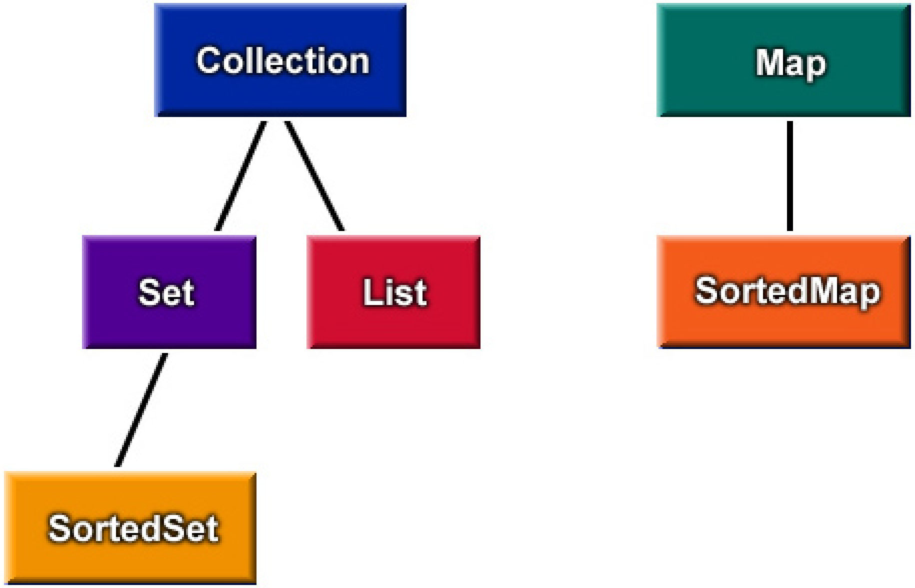
Design Goals

- 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<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);         // Optional
    boolean remove(Object element); // Optional
    Iterator<E> iterator();

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

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? Extends E> 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<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
}
```

Collection Example

Reusable algorithm to eliminate nulls

```
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<E> extends Collection<E> {
}
```

Set Idioms

```
Set<Type> s1, s2;

boolean isSubset = s1.containsAll(s2);

Set<Type> union = new HashSet<>(s1);
union = union.addAll(s2);

Set<Type> intersection = new HashSet<>(s1);
intersection.retainAll(s2);

Set<Type> difference = new HashSet<>(s1);
difference.removeAll(s2);

Collection<Type> c;
Collection<Type> noDups = new HashSet<>(c);
```

`List` Interface

A sequence of objects

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

    List<E> subList(int from, int to);

    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
}
```

List Example

Reusable algorithms to swap and randomize

```
public static <E> void swap(List<E> a, int i, int j) {
    E 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<Type> a, b;

// Concatenate two lists
a.addAll(b);

// Range-remove
a.subList(from, to).clear();

// Range-extract
List<Type> partView = a.subList(from, to);
List<Type> part = new ArrayList<>(partView);
partView.clear();
```

Map Interface

A key-value mapping

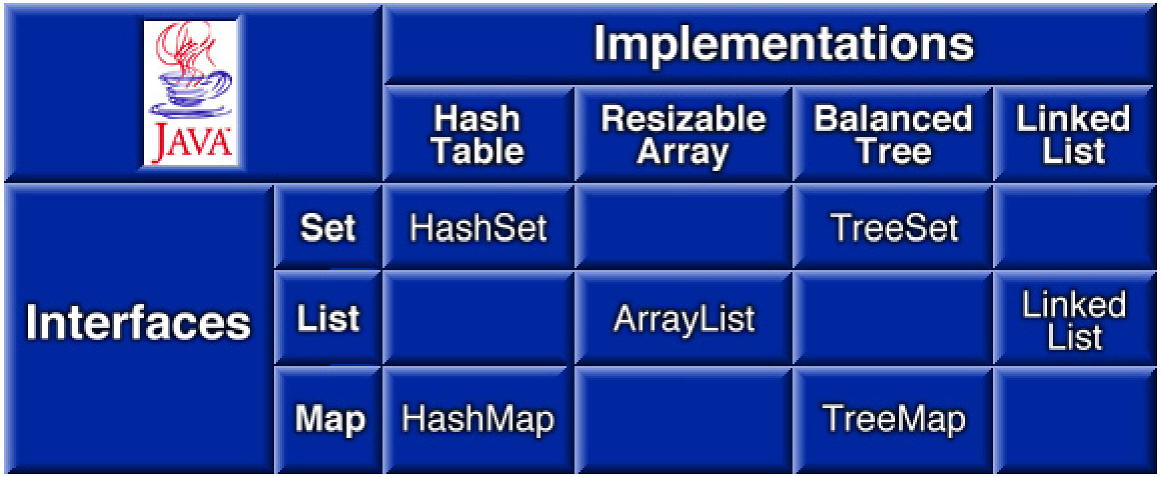
```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    Object get(Object key);
    Object put(K key, V value);   // Optional
    Object remove(Object key);    // Optional
    void putAll(Map<? Extends K, ? Extends V> t); // Opt.
    void clear();        // Optional

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
}
```

Map Idioms

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

// Iterate over all keys in Map m as of Java 5 (2004)
for (Key k : m.keySet())
    System.out.println(i.next());

// "Map algebra"
Map<Key, Val> a, b;
boolean isSubMap = a.entrySet().containsAll(b.entrySet());

Set<Key> commonKeys =
    new HashSet<>(a.keySet()).retainAll(b.keySet); [sic!]

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

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 (No longer a good idea; use ArrayDeque instead.)

Implementation Behavior

Unlike `Vector` *and* `Hashtable`…

- 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
- It was new then; it's old now!
  - Synchronization wrappers are now largely obsolete
  - Made obsolete by **concurrent collections**

Synchronization Wrapper Example

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

// In Java 5 (post-2004)
synchronized(s) {
    for (String t : s)
        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(E[] a)`
  - Allows array to be "viewed" as List
  - Bridge to Collection-based APIs
- `EMPTY_SET`, `EMPTY_LIST`, `EMPTY_MAP`
  - immutable constants
- `singleton(E o)`
  - immutable set with specified object
- `nCopies(E 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(int[] a) {
    return new AbstractList() {
        public Integer get(int i) {
            return new Integer(a[i]);
        }

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

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

Reusable Algorithms

```
static <T extends Comparable<? super T>> void sort(List<T> list);
static int binarySearch(List list, Object key);
static <T extends Comparable<? super T>> T min(Collection<T> coll);
static <T extends Comparable<? super T>> T max(Collection<T> coll);
static <E> void fill(List<E> list, E e);
static <E> void copy(List<E> dest, List<? Extends E> src);
static void reverse(List<?> list);
static void shuffle(List<?> list);
```

Algorithm Example 1

Sorting lists of comparable elements

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


LinkedList<Date> dates;    // Elements type: Date
    ...
Collections.sort(dates);   // Chronological order


// Comparable interface (Infrastructure)
public interface Comparable<E extends Comparable<E>> {
    int compareTo(Object o);
}
```

Comparator Interface

Infrastructure

- Specifies order among objects
  - Overrides natural order on comparables
  - Provides order on non-comparables

```
public interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

Algorithm Example 2

Sorting with a comparator

```
List<String> strings; // Element type: String

Collections.sort(strings, Collections.reverseOrder());


// Case-independent alphabetical order
static Comparator<String> cia = new Comparator<>() {
    public int compare(String c1, String c2) {
        return c1.toLowerCase().compareTo(c2.toLowerCase());
    }
};


Collections.sort(strings, cia);
```

Compatibility

Old and new collections interoperate freely

- Upward Compatibility
  - `Vector<E> implements List<E>`
  - `Hashtable<K,V> implements Map<K,V>`
  - `Arrays.asList(myArray)`
- Backward Compatibility
  - `myCollection.toArray()`
  - `new Vector<>(myCollection)`
  - `new Hashtable<>(myMap)`

API Design Guidelines

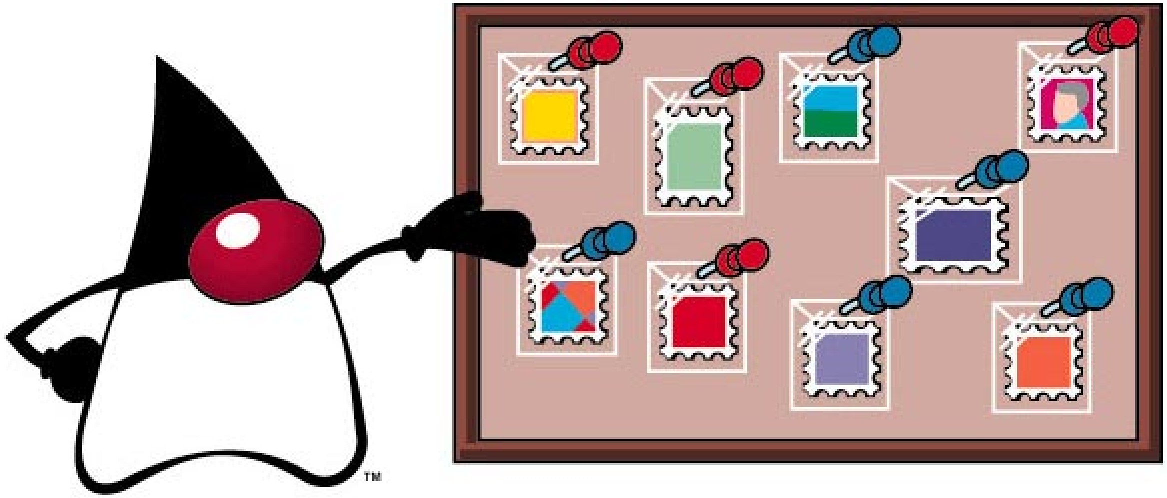
- Avoid ad hoc collections
  - Input parameter type:
    - Any collection **interface** (`Collection`, Map best)
    - Array may sometimes be preferable
  - Output value type:
    - Any collection **interface** or **class**
    - Array
- Provide adapters for your legacy collections

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
  - With arguable exception of Java 8 streams & default methods on core interfaces (2014)

Part 2: Outline

I. The initial release of the collections API

II. Design of the first release

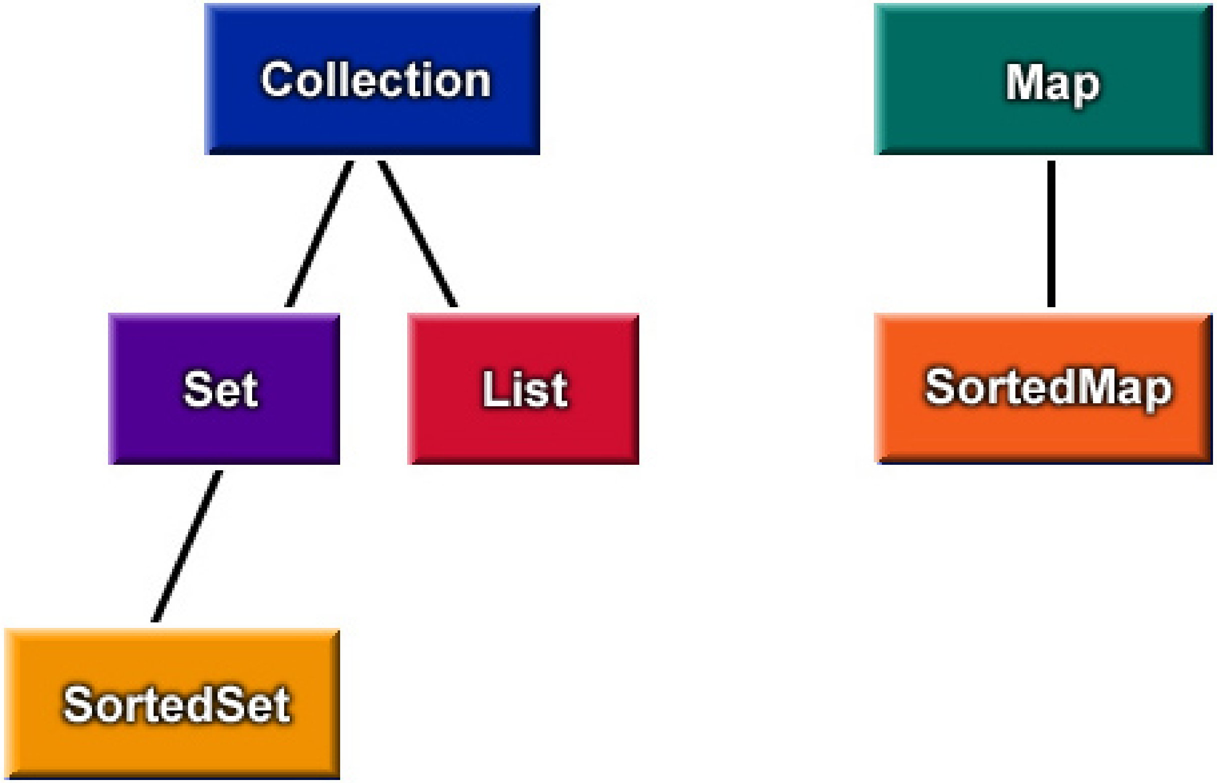
III. Evolution

IV. Code example

V. Critique

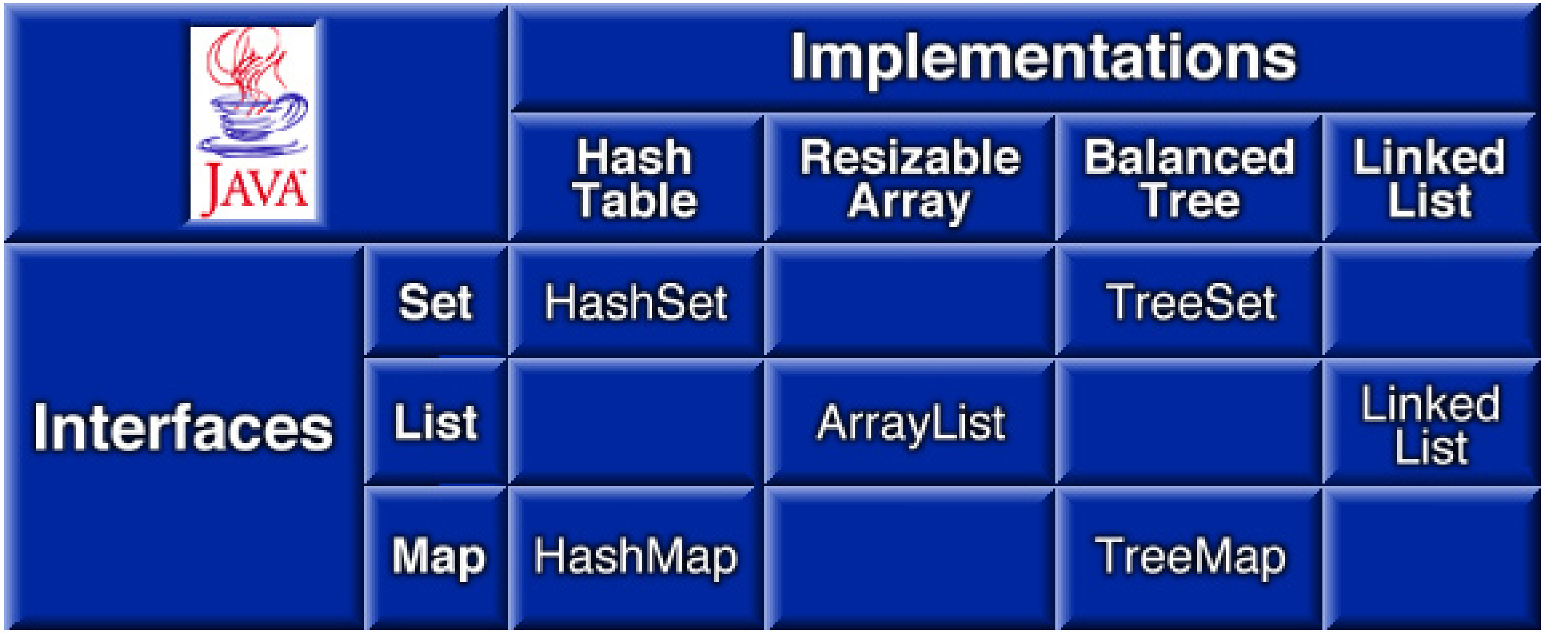
Collection **interfaces**

First release, 1998



General-purpose **implementations**

First release, 1998



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

Other **implementations**

First release, 1998

- Convenience implementations
  - `Arrays.asList(Object[] a)`
  - `EMPTY_SET, EMPTY_LIST, EMPTY_MAP`
  - `singleton(Object o)`
  - `nCopies(Object o)`
- Decorator implementations
  - `Unmodifiable{Collection,Set,List,Map,SortedMap}`
  - `Synchronized{Collection,Set,List,Map,SortedMap}`
- Special Purpose implementation – `WeakHashMap`

Reusable **algorithms**

First release, 1998

- `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);`

And that’s all there was to it!

OK, I told a little white lie: there were also array utilities

First release, 1998

- static int binarySearch(*type*[] a, *type* key)
- static int binarySearch(Object[] a, Object key, Comparator c)
- static boolean equals(*type*[] a, *type*[] a2)
- static void fill(*type*[] a, *type* val)
- static void fill(*type*[] a, int fromIndex, int toIndex, *type* val)
- static void sort(*type*[] a)
- static void sort(*type*[] a, int fromIndex, int toIndex)
- static void sort(*type*[] a, Comparator c)
- static void sort(*type*[] a, int fromIdx, int toidx, Comparator c)

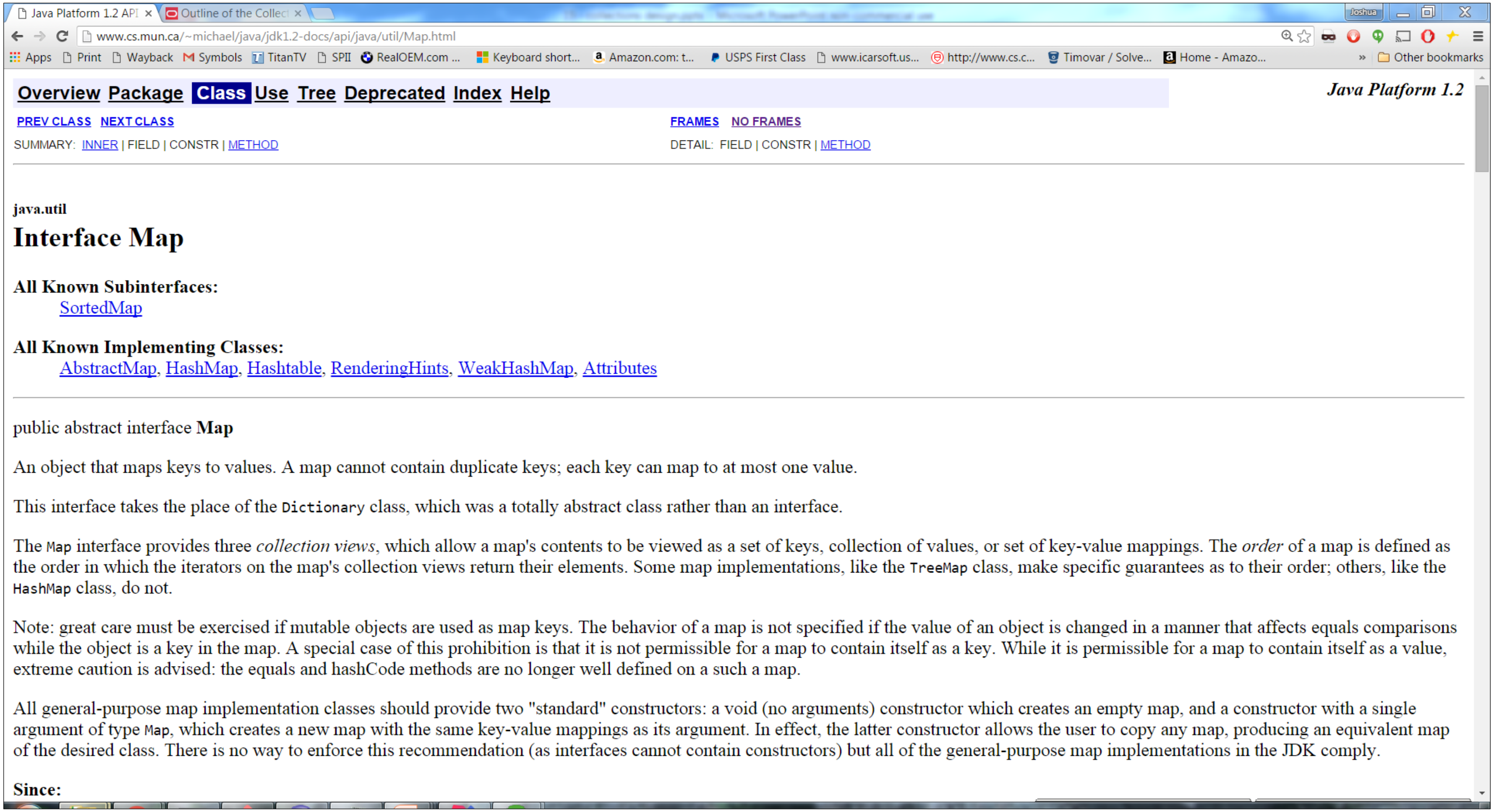
Documentation matters

"Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation."

-D. L. Parnas, 1994;
In Brooks, *The Mythical Man Month*

Of course you need good JavaDoc

But it is not sufficient for a substantial API!



Overview Package **Class** Use Tree Deprecated Index Help

Java Platform 1.2

PREV CLASS NEXT CLASS

FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

java.util

Interface Map

All Known Subinterfaces:

SortedMap

All Known Implementing Classes:

AbstractMap, HashMap, Hashtable, RenderingHints, WeakHashMap, Attributes

public abstract interface **Map**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

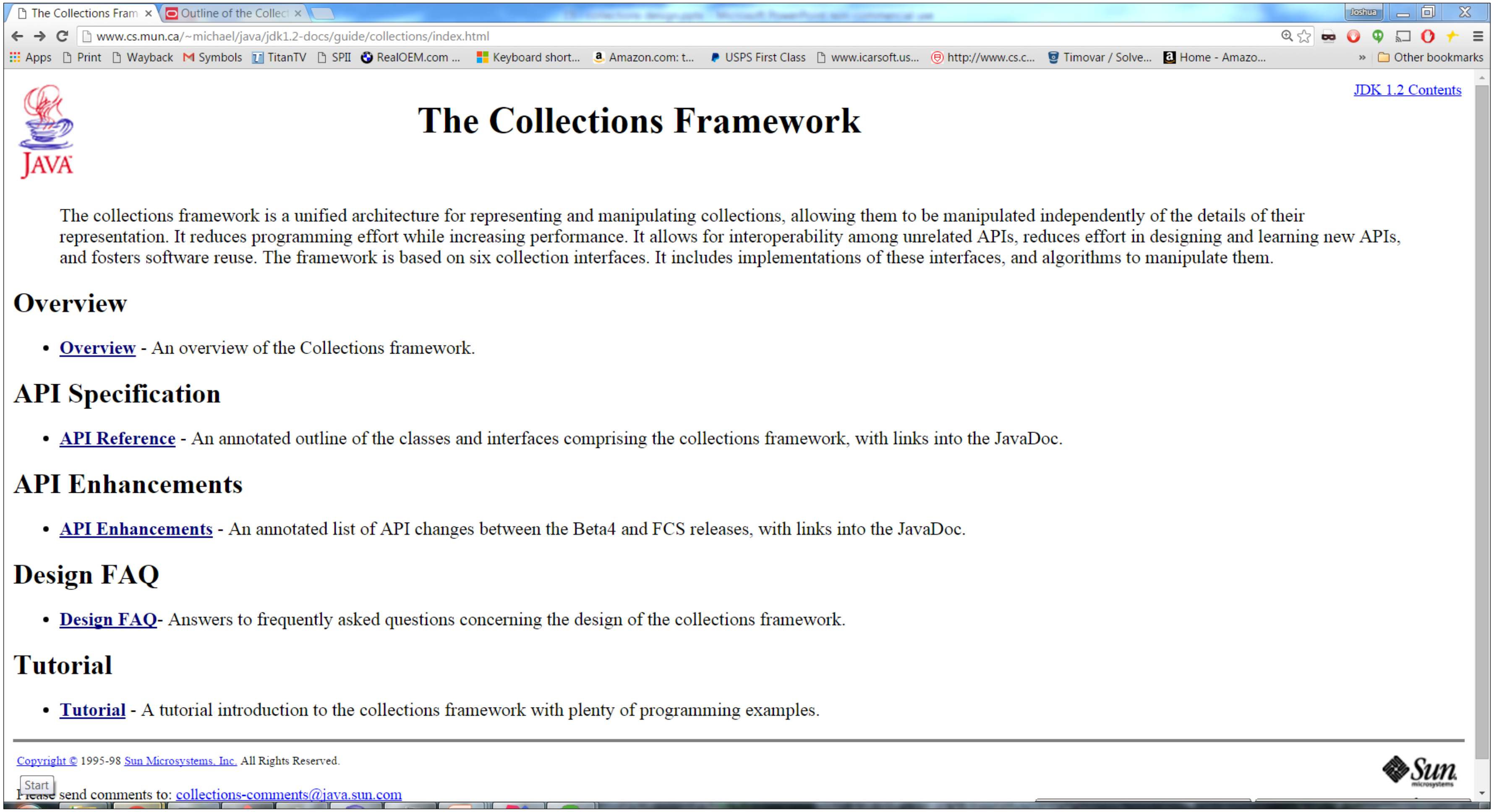
The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on a such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

Since:

A single place to go for documentation



JDK 1.2 Contents

The Collections Framework

The collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse. The framework is based on six collection interfaces. It includes implementations of these interfaces, and algorithms to manipulate them.

Overview

- **Overview** - An overview of the Collections framework.

API Specification

- **API Reference** - An annotated outline of the classes and interfaces comprising the collections framework, with links into the JavaDoc.

API Enhancements

- **API Enhancements** - An annotated list of API changes between the Beta4 and FCS releases, with links into the JavaDoc.

Design FAQ

- **Design FAQ**- Answers to frequently asked questions concerning the design of the collections framework.

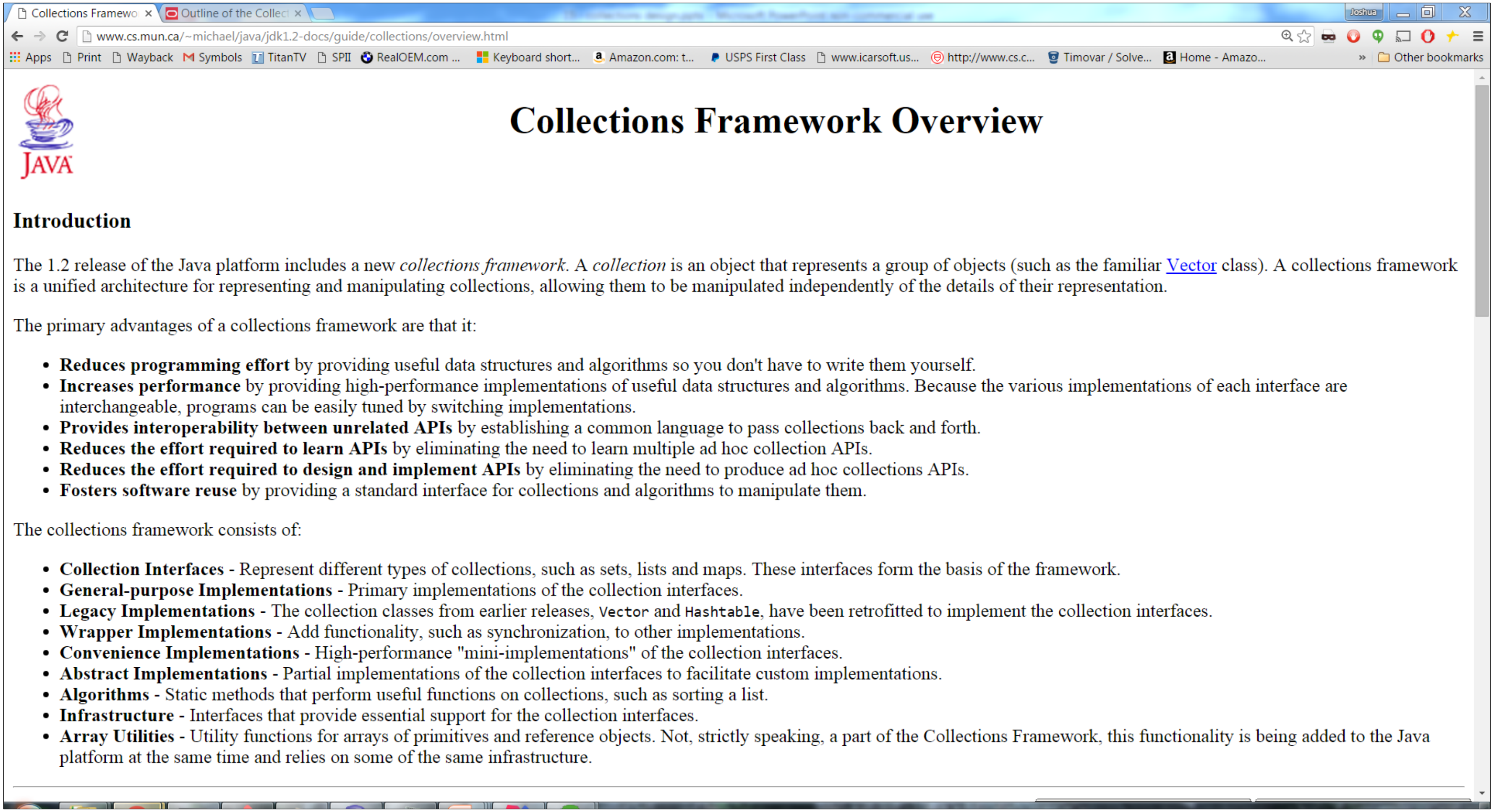
Tutorial

- **Tutorial** - A tutorial introduction to the collections framework with plenty of programming examples.

Please send comments to: collections-comments@java.sun.com

Overviews provide understanding

A place to go when first learning an API



Collections Framewo ×
Outline of the Collect ×
www.cs.mun.ca/~michael/java/jdk1.2-docs/guide/collections/overview.html
Apps Print Wayback Symbols TitanTV SPII RealOEM.com ... Keyboard short... Amazon.com: t... USPS First Class www.icarsoft.us... http://www.cs.c... Timovar / Solve... Home - Amazo... Other bookmarks
JAVA

Collections Framework Overview

Introduction

The 1.2 release of the Java platform includes a new *collections framework*. A *collection* is an object that represents a group of objects (such as the familiar Vector class). A collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.

The primary advantages of a collections framework are that it:

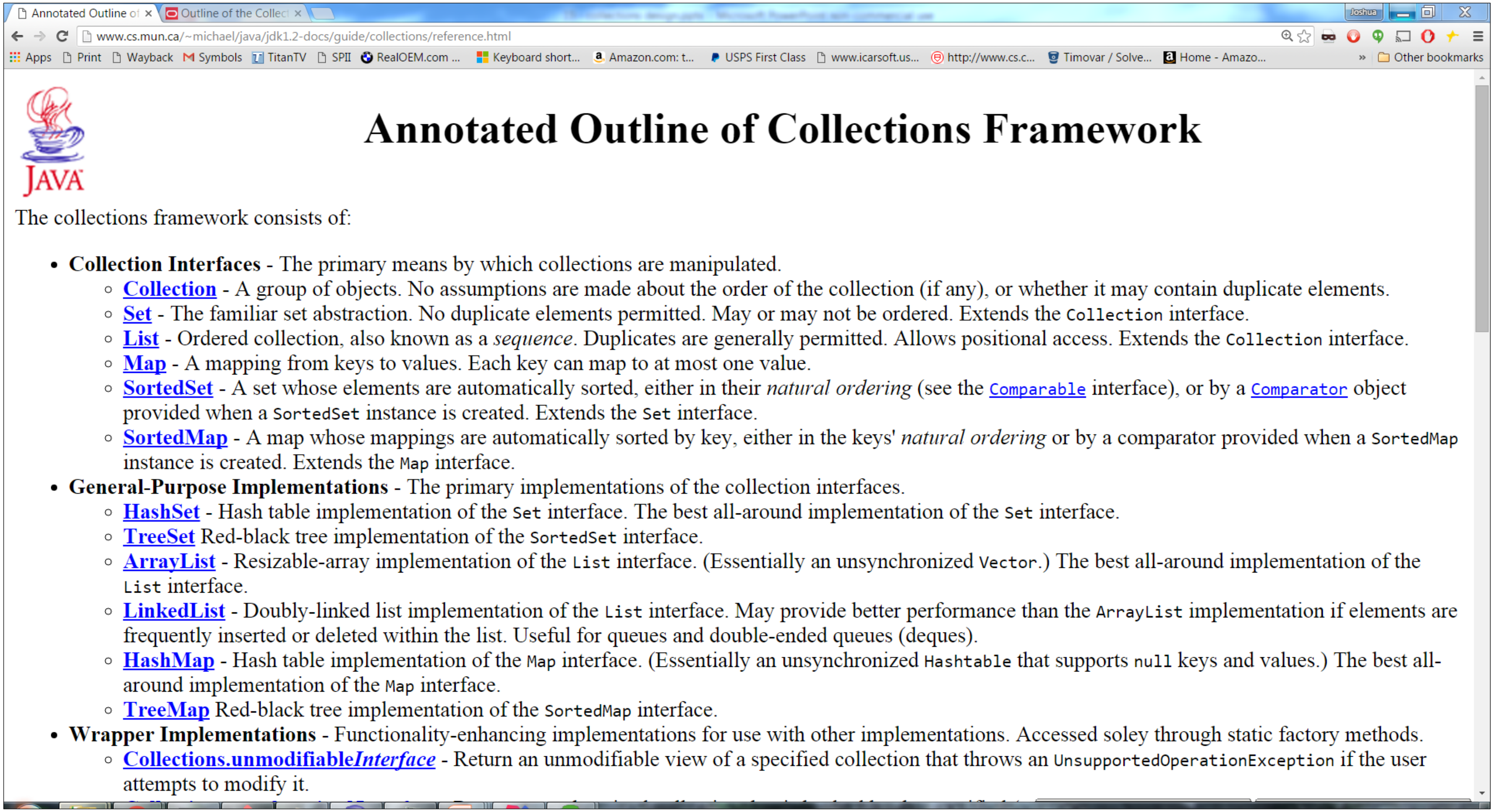
- **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

The collections framework consists of:

- **Collection Interfaces** - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- **General-purpose Implementations** - Primary implementations of the collection interfaces.
- **Legacy Implementations** - The collection classes from earlier releases, `Vector` and `Hashtable`, have been retrofitted to implement the collection interfaces.
- **Wrapper Implementations** - Add functionality, such as synchronization, to other implementations.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.
- **Abstract Implementations** - Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms** - Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure** - Interfaces that provide essential support for the collection interfaces.
- **Array Utilities** - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality is being added to the Java platform at the same time and relies on some of the same infrastructure.

Annotated outlines provide access

They're awesome and underutilized



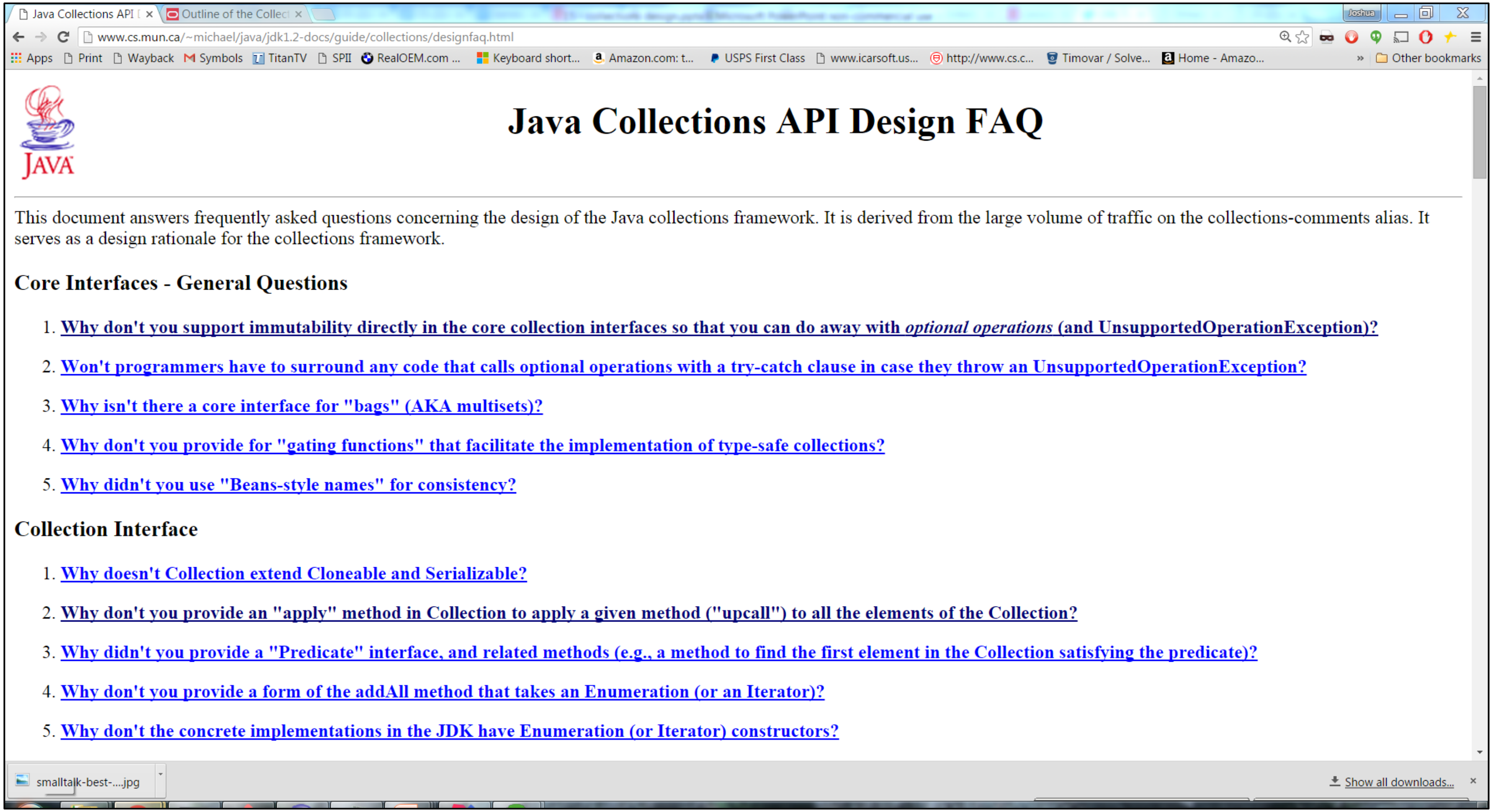
Annotated Outline of Collections Framework

The collections framework consists of:

- **Collection Interfaces** - The primary means by which collections are manipulated.
  - Collection - A group of objects. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements.
  - Set - The familiar set abstraction. No duplicate elements permitted. May or may not be ordered. Extends the `Collection` interface.
  - List - Ordered collection, also known as a *sequence*. Duplicates are generally permitted. Allows positional access. Extends the `Collection` interface.
  - Map - A mapping from keys to values. Each key can map to at most one value.
  - SortedSet - A set whose elements are automatically sorted, either in their *natural ordering* (see the `Comparable` interface), or by a `Comparator` object provided when a `SortedSet` instance is created. Extends the `Set` interface.
  - SortedMap - A map whose mappings are automatically sorted by key, either in the keys' *natural ordering* or by a comparator provided when a `SortedMap` instance is created. Extends the `Map` interface.
- **General-Purpose Implementations** - The primary implementations of the collection interfaces.
  - HashSet - Hash table implementation of the `Set` interface. The best all-around implementation of the `Set` interface.
  - TreeSet Red-black tree implementation of the `SortedSet` interface.
  - ArrayList - Resizable-array implementation of the `List` interface. (Essentially an unsynchronized `Vector`.) The best all-around implementation of the `List` interface.
  - LinkedList - Doubly-linked list implementation of the `List` interface. May provide better performance than the `ArrayList` implementation if elements are frequently inserted or deleted within the list. Useful for queues and double-ended queues (deques).
  - HashMap - Hash table implementation of the `Map` interface. (Essentially an unsynchronized `Hashtable` that supports `null` keys and values.) The best all-around implementation of the `Map` interface.
  - TreeMap Red-black tree implementation of the `SortedMap` interface.
- **Wrapper Implementations** - Functionality-enhancing implementations for use with other implementations. Accessed soley through static factory methods.
  - Collections.unmodifiable*Interface* - Return an unmodifiable view of a specified collection that throws an `UnsupportedOperationException` if the user attempts to modify it.

A design rationale saves you hassle

and provides a testament to history ("apologia")



Java Collections API Design FAQ

This document answers frequently asked questions concerning the design of the Java collections framework. It is derived from the large volume of traffic on the collections-comments alias. It serves as a design rationale for the collections framework.

Core Interfaces - General Questions

1. Why don't you support immutability directly in the core collection interfaces so that you can do away with *optional operations* (and UnsupportedOperationException)?
2. Won't programmers have to surround any code that calls optional operations with a try-catch clause in case they throw an UnsupportedOperationException?
3. Why isn't there a core interface for "bags" (AKA multisets)?
4. Why don't you provide for "gating functions" that facilitate the implementation of type-safe collections?
5. Why didn't you use "Beans-style names" for consistency?

Collection Interface

1. Why doesn't Collection extend Cloneable and Serializable?
2. Why don't you provide an "apply" method in Collection to apply a given method ("upcall") to all the elements of the Collection?
3. Why didn't you provide a "Predicate" interface, and related methods (e.g., a method to find the first element in the Collection satisfying the predicate)?
4. Why don't you provide a form of the addAll method that takes an Enumeration (or an Iterator)?
5. Why don't the concrete implementations in the JDK have Enumeration (or Iterator) constructors?

Outline

I. The initial release of the collections API

II. Design of the first release

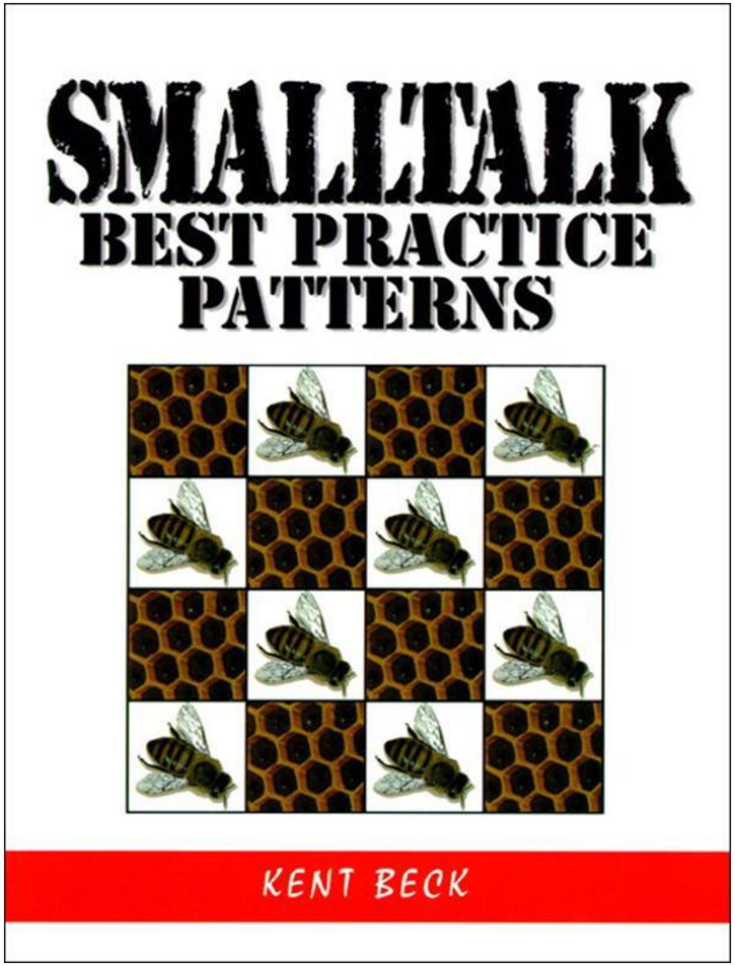
III. Evolution

IV. Code example

V. Critique

A wonderful source of use cases

“Good artists copy, great artists steal.” – Pablo Picasso (ascribed)



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
 * <i>views</i> on some backing collection.  (It is necessary to
 * implement this interface only if the backing collection is not
 * <i>encapsulated</i> by this Collection or Table; that is, if the
 * backing collection might conceivably be be accessed in some way other
 * than through this Collection or Table.)  This allows users
 * to detect potential <i>aliasing</i> between collections.
 * <p>
 * If a user attempts to modify one collection
 * object 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 possiblity of <i>aliasing</i> 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 whenver the possibility of
     * aliasing exists.
     * <p>
     * If this collection is a view on an Object that does not impelement
     * Alias, this method must return the IdentityHashCode of the backing
     * Object.  For example, a List backed by a user-provided array would
     * return the IdentityHashCode of the array.
     * If this collection is a <i>view</i> 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).
     * <p>
     * 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.
     * <p>
     * For any collection backed by multiple Objects (a "concatenation
     * view" of two Lists, for instance), this method must return zero.
     * Similarly, for any <i>view</i> 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.
     *  <p>
     * The possibility of aliasing between two collections exists iff
     * any of the following conditions are true:<ol>
     *            <li>The two collections are the same Object.
     *            <li>Either collection implements Alias and has a
     *                backingObjectId that is the identityHashCode of
     *                the other collection.
     *            <li>Either collection implements Alias and has a
     *                backingObjectId of zero.
     *            <li>Both collections implement Alias and they have equal
     *                backingObjectId's.</ol>
     *
     * @see java.lang.System#identityHashCode
     * @since JDK1.2
     */
    int backingObjectId();
}
```

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

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

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

Simiarly, I don't think that the table Entry subclass or the various views mechanisms carry their weight.

III. Evolution of Java collections

Release, Year	Changes
JDK 1.0, 1996	Java Released: `Vector`, `Hashtable`, `Enumeration`
JDK 1.1, 1996	(No API changes)
J2SE 1.2, 1998	Collections framework added
J2SE 1.3, 2000	(No API changes)
J2SE 1.4, 2002	`LinkedHash{Map,Set}`, `IdentityHashSet`, 6 new algorithms
J2SE 5.0, 2004	Generics, for-each, enums: generified everything, `Iterable`, `Queue`, `Enum{Set,Map}`, concurrent collections
Java 6, 2006	`Deque`, `Navigable{Set,Map}`, `newSetFromMap`, `asLifoQueue`
Java 7, 2011	No API changes. Improved sorts & defensive hashing
Java 8, 2014	Lambdas, streams, new methods on core interfaces
Java 9, 2017	Convenience factories: `Set.of()`, `List.of`, `Map.of()`

IV. Example – how to find anagrams

- Alphabetize the characters in each word
  - cat → act, dog → dgo, mouse → emosu
  - Resulting string is called *alphagram*
- **Anagrams share the same alphagram!**
  - stop → opst, post → opst, tops → opst, opts → opst
- So go through words making “multimap”from alphagram to word

How to find anagrams in Java (1/2)

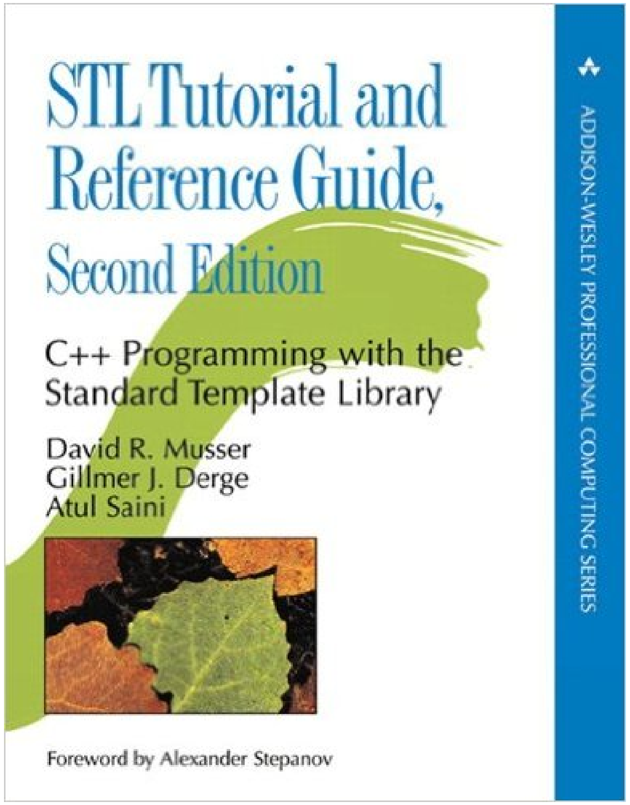
```java
public static void main(String[] args) throws IOException {
    // Read words from file and put into a simulated multimap
    Map<String, List<String>> groups = new HashMap<>();
    try (Scanner s = new Scanner(new File(args[0]))) {
        while (s.hasNext()) {
            String word = s.next();
            String alpha = alphabetize(word);

            // Next 4 lines could be replaced by merge method (Java 8)
            List<String> group = groups.get(alpha);
            if (group == null)
                groups.put(alpha, group = new ArrayList<>());
            group.add(word);
        }
    }
```

How to find anagrams in Java (2/2)

```
    // Print all anagram groups above size threshold
    int minGroupSize = Integer.parseInt(args[1]);
    for (List<String> group : groups.values())
        if (group.size() >= minGroupSize)
            System.out.println(group.size() + ": " + group);

    // Returns the alphagram for a string
    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

Two slides in Java vs. **an entire chapter** in STL

Java's verbosity is somewhat exaggerated



STL Tutorial and Reference Guide, Second Edition

C++ Programming with the Standard Template Library

David R. Musser
Gillmer J. Derge
Atul Saini

Foreword by Alexander Stepanov

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Epilogue: How to find anagrams in Java 8 (streams)

```
public class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(groupingBy(word -> alphabetize(word)))
                .values().stream()
                .filter(group -> group.size() >= minGroupSize)
                .forEach(g -> System.out.println(g.size() + ": " + g));
        }
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

V. Critique

Some things I wish I'd done differently

- Algorithms should return collection, not `void` or `boolean`
  - Enables **fluent APIs**, turns ugly multiliners into nice one-liners

```
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 had proper navigation
  - Navigable{Set,Map} turned Sorted{Set,Map} into warts

Conclusion

- **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