The Collections API

Mark Allen Weiss

Copyright 2000

Lecture Objectives

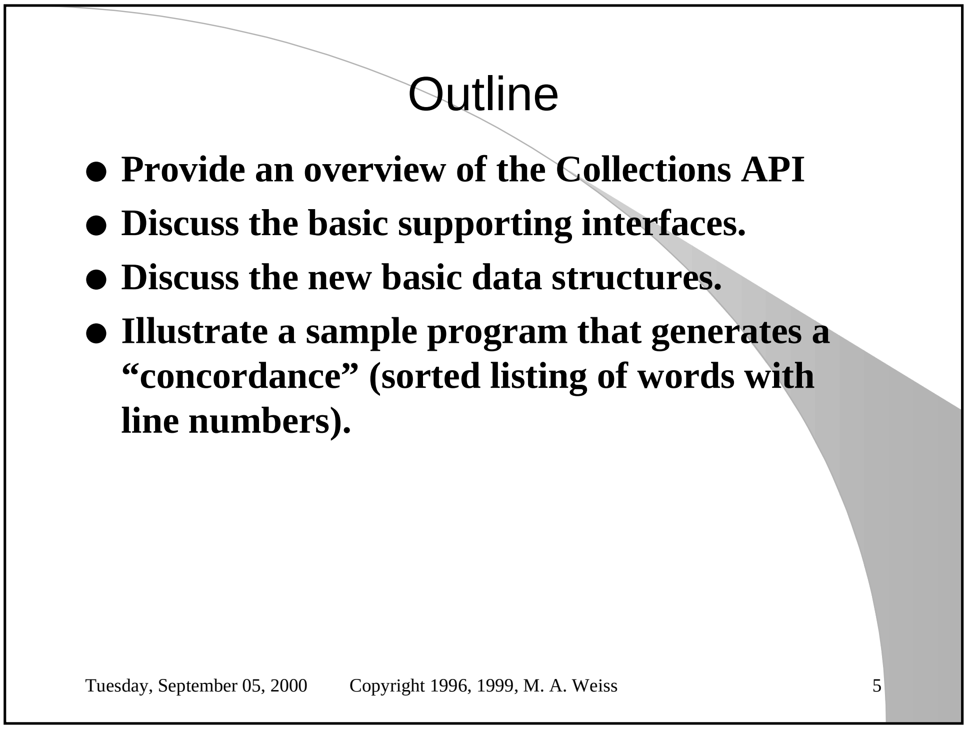
- **To see some bad design (Java 1.1)**
- **To see a better design (Java 1.2)**
- **To learn how to use the Collections package in Java 1.2.**
- **To illustrate features of Java that help (and hurt) the design of the Collections API.**

The Collections API in Java 1.1

- Basically four classes plus one interface:
  - `Vector` (resizeable generic array)
  - `Stack`
  - `Hashtable` (map of keys and values)
  - `Properties` (map of keys and values that are Strings)
  - `Enumeration` (a sloppy iterator pattern)
- Pathetic Design
  - `Stack` IS-A `Vector`?
  - `Properties` IS-A `Hashtable`?

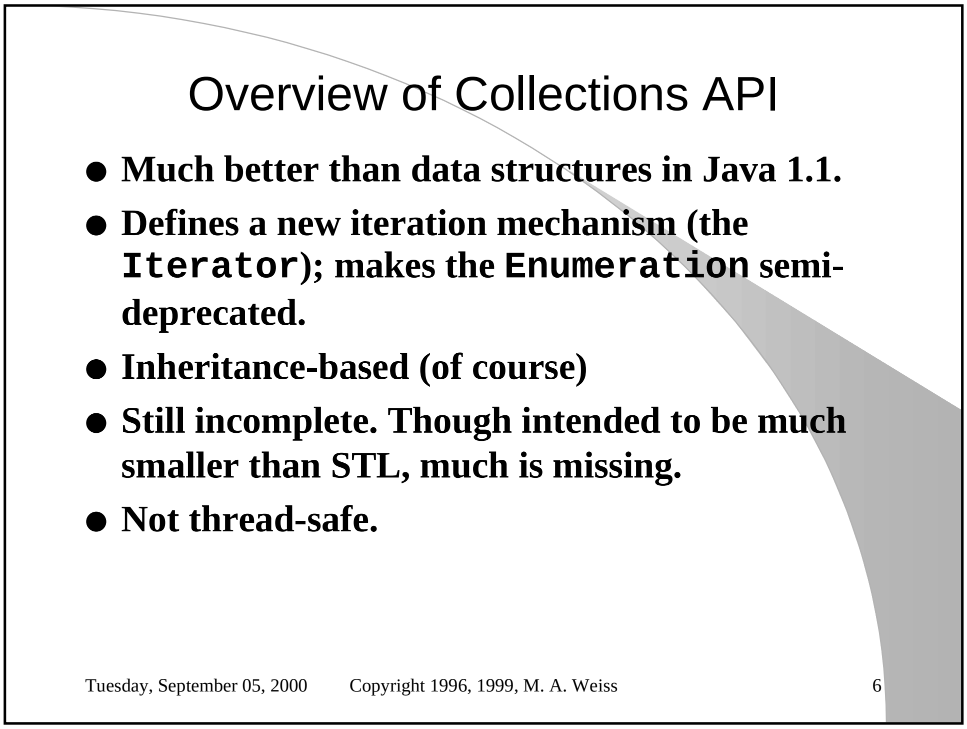
The Collections API In Java 1.2

- Deprecates the Java 1.1 stuff
- Contains new data structures including linked list, queue, set, and map.
- Contains generic algorithms including sorting.
- Mostly in `java.util`.



Outline

- Provide an overview of the Collections API
- Discuss the basic supporting interfaces.
- Discuss the new basic data structures.
- Illustrate a sample program that generates a "concordance" (sorted listing of words with line numbers).



Overview of Collections API

- Much better than data structures in Java 1.1.
- Defines a new iteration mechanism (the `Iterator`); makes the `Enumeration` semi-deprecated.
- Inheritance-based (of course)
- Still incomplete. Though intended to be much smaller than STL, much is missing.
- Not thread-safe.

Basic Supporting Interfaces

- **There are some new supporting interfaces. The four most important are:**
  - **`Collection`**
  - **`Iterator`**
  - **`Comparable`**
  - **`Comparator`**

`Collection` Interface

- **Represents a group of objects (its *elements*)**
- **Different implementations place restrictions (such as allowing/disallowing duplicates, maintaining the collection in sorted order)**
- **Basic operations:**

```
        boolean contains( Object element )
        boolean isEmpty( )
        int size( )
        Iterator iterator( )
```

- **To design your own implementation of a `Collection`, extend `AbstractCollection`.**

More On Collection

- **All collections, by convention, have two constructors:**
  - **Construct empty**
  - **Construct with a set of references that reference objects in any other collection**
- **`AbstractCollection` is an abstract class that implements many of the “generic” methods in the `Collection` interface.**

`Iterator` Interface

- **Provides three methods that are used to access any `Collection`.**

```
boolean hasNext( )
Object next( )
void remove( )
```

- **`hasNext` returns `true` if the iteration has more items. `next` returns the next item and advances the iterator. `remove` removes the last accessed item (can’t be called twice in a row).**
- **Officially preferred over `Enumeration`.**
- **Not a great iterator pattern because advancing and accessing current item are combined.**

Example

- **Output the contents of *any* `Collection`.**

```
static void printCollection( Collection C )
{
    Iterator itr = C.iterator( );
    while( itr.hasNext( ) )
        System.out.println( itr.next( ) );
}
```

- **If the underlying collection is sorted, the output will be sorted.**
- **Not bidirectional (but other iterators are).**
- **There are no public concrete iterators!!**

How Do You Get An Iterator?

- **Each `Collection` class defines a concrete class that implements the `Iterator` interface**
  - **`ArrayList` could define `ArrayListIterator`**
  - **`TreeSet` could define `TreeSetIterator`**
- **The `iterator()` method creates an instance of the appropriate concrete class and returns it.**
- **Static type of the return is `Iterator`.**
- **Dynamic type is the concrete `Iterator`.**
- **Could make the concrete implementation of `Iterator` package-visible and hide it.**

`Comparable` Interface

- **Defined in `java.lang`. Has one method:**

```
    int compareTo( Object rhs )
                throws ClassCastException
```

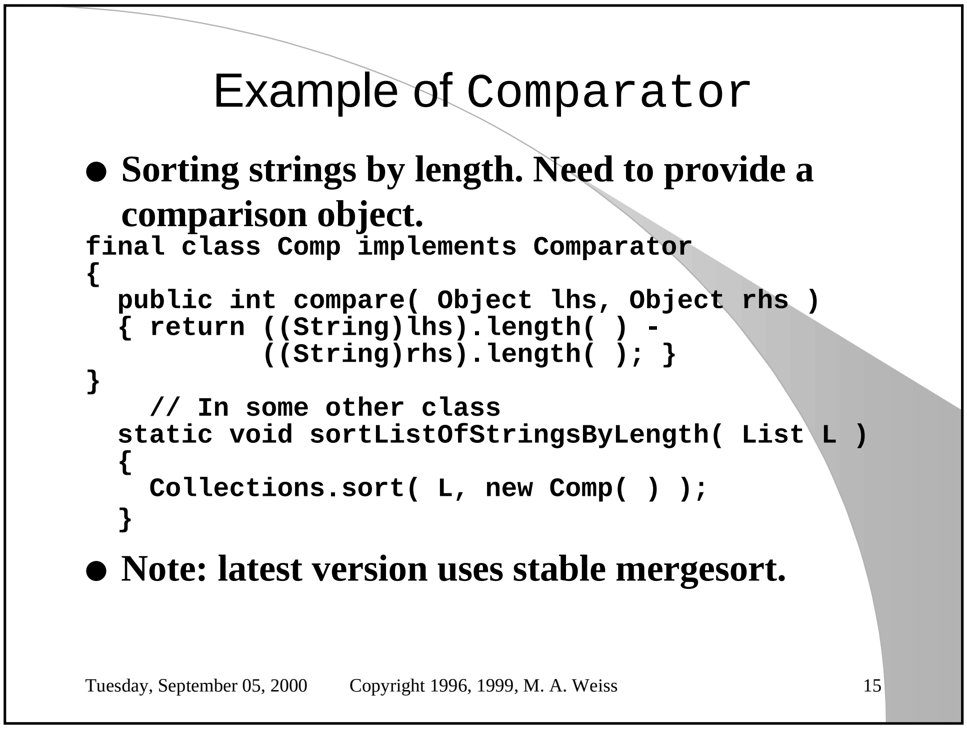
- **Same semantics as `String`. `String` implements `Comparable`, as do the primitive wrapper classes (e.g. `Integer`).**
- **If you have a `Comparable` class in your code, you may have a conflict in Java 1.2.**

`Comparator` Interface

- **Has one method:**

```
    int compares( Object lhs, Object rhs )
```

- **Compares two objects, with return value that is like `compareTo`.**
- **Use to override the default (or non-existent ordering) for collections that are sorted.**
- **Similar to the function object in STL.**
- **Predefined constant function object is `Collections.REVERSE_ORDER`.**

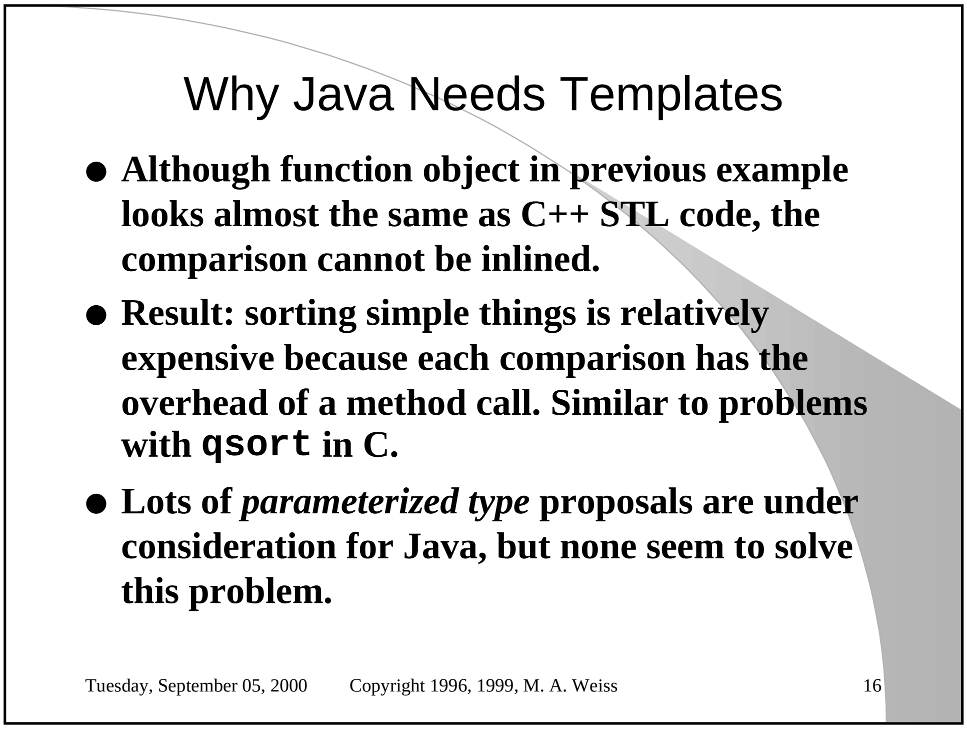


Example of `Comparator`

- **Sorting strings by length. Need to provide a comparison object.**

```
final class Comp implements Comparator
{
  public int compare( Object lhs, Object rhs )
  { return ((String)lhs).length( ) -
           ((String)rhs).length( ); }
}
    // In some other class
  static void sortListOfStringsByLength( List L )
  {
    Collections.sort( L, new Comp( ) );
  }
```

- **Note: latest version uses stable mergesort.**



Why Java Needs Templates

- **Although function object in previous example looks almost the same as C++ STL code, the comparison cannot be inlined.**
- **Result: sorting simple things is relatively expensive because each comparison has the overhead of a method call. Similar to problems with `qsort` in C.**
- **Lots of *parameterized type* proposals are under consideration for Java, but none seem to solve this problem.**

Data Structures

- **Several data structures**
  - **List, with list iterator**
  - **Stack and Queue**
  - **Set**
  - **Map**
- **Not thread-safe.**

`List`

- **Ordered collection (also known as *sequence*). Position in the list matters and can be specified by an integer index (0 is first position). Elements are not necessarily sorted.**
- **`List` is an interface. It is implemented by `ArrayList`, `LinkedList` (also `Vector`).**
- **Watch out for `java.awt.List` conflict.**

`ArrayList` and `Vector`

- **Useful if you need to access by position, because you can do direct indexing.**
- **Insertions and deletions are expensive, except at high-end.**
- **Insertion at the end of an `ArrayList` causes an expansion if full with a guarantee of efficient performance.**
- **`ArrayList` is preferred over `Vector`.**
- **`Vector` is retrofitted to implement `List` interface. Useful if thread-safety is needed.**

`LinkedList` methods

- **Implements a doubly-linked `List`.**
- **Lots of methods. Here are some:**

```
void addFirst( )
void addLast( )
Object getFirst( )
Object getLast( )
Object removeFirst( )
Object removeLast( )
void clear( )
ListIterator listIterator( int index )
```

- **Can implement stack and queue operations.**
- **Access with `get` and `set` supported but obviously horrendously slow.**

`List` (Continued)

- **`ListIterator` is an interface that supports bi-directional iteration. Also (optionally) supports `add` (insert a new element prior to the next element in the iteration) and `remove` (removes last accessed element)**
- **`Stack` class from Java 1.1 is still here, but is synchronized and could be slow.**
- **There is no class named `Queue`.**

Using The `List` Interface Type

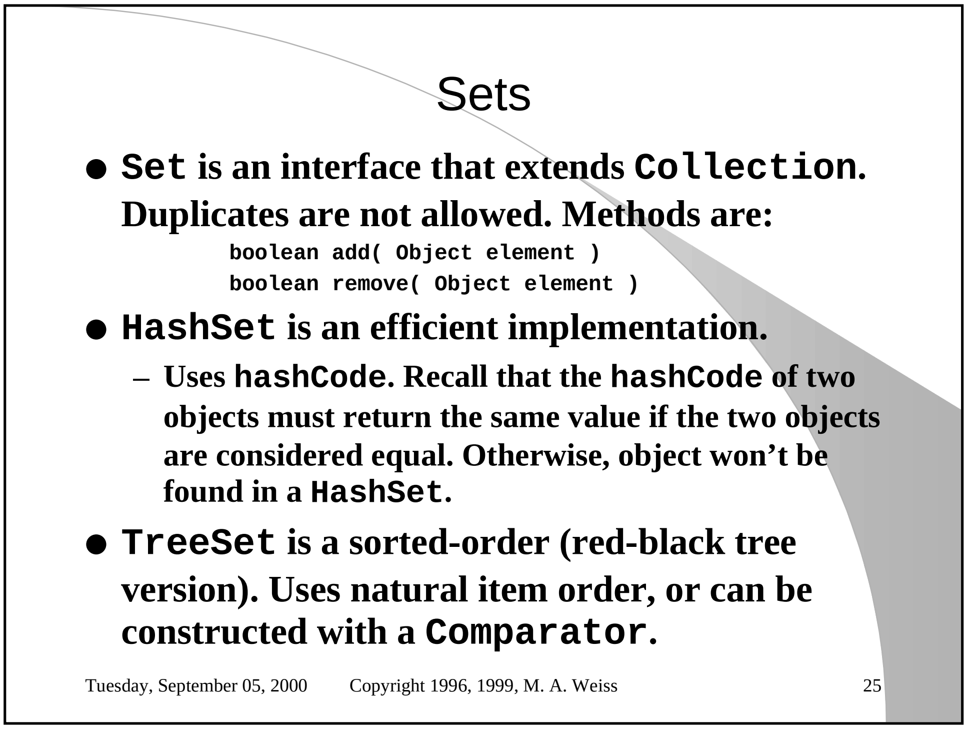
- **If only `ArrayList` or `LinkedList` operations you are using are defined in `List` interface, should declare the reference using the `List` interface.**
  - **Makes code more flexible**
  - **Can change implementation from `ArrayList` to `LinkedList` later**
  - **Same idea of preferring `Reader`/`Writer` as reference types**

Optional Methods

- **Starting in Java 1.2, interfaces can specify that some of its methods are “optional.”**
- **Implementor will throw `UnsupportedOperationException` if it does not want to implement an optional method. This is a runtime exception.**
- **Purely a convention; no language rule involved.**
- **Useful if you are**
  - **lazy; or**
  - **implementing immutable containers**

More On Optional Methods

- **Convention is that interface will document that the method might not be supported.**
- **Caller is expected to check documentation of class that implements the interface to see if method is supported.**
- **If caller doesn’t do that, and calls the method anyway, will get an exception. Clearly this is considered a programming error, so it is a runtime exception.**
- **Optional methods are somewhat controversial.**

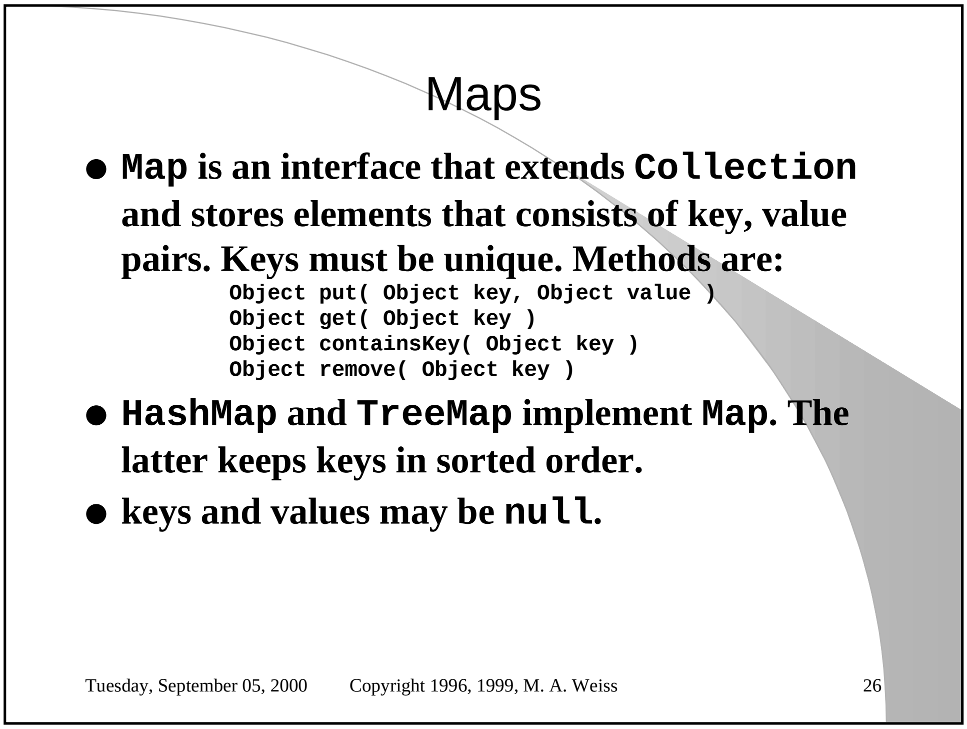


Sets

- **`Set` is an interface that extends `Collection`. Duplicates are not allowed. Methods are:**

```
boolean add( Object element )
boolean remove( Object element )
```

- **`HashSet` is an efficient implementation.**
  - **Uses `hashCode`. Recall that the `hashCode` of two objects must return the same value if the two objects are considered equal. Otherwise, object won't be found in a `HashSet`.**
- **`TreeSet` is a sorted-order (red-black tree version). Uses natural item order, or can be constructed with a `Comparator`.**



Maps

- **`Map` is an interface that extends `Collection` and stores elements that consists of key, value pairs. Keys must be unique. Methods are:**

```
Object put( Object key, Object value )
Object get( Object key )
Object containsKey( Object key )
Object remove( Object key )
```

- **`HashMap` and `TreeMap` implement `Map`. The latter keeps keys in sorted order.**
- **keys and values may be `null`.**

Getting a Collection from a Map

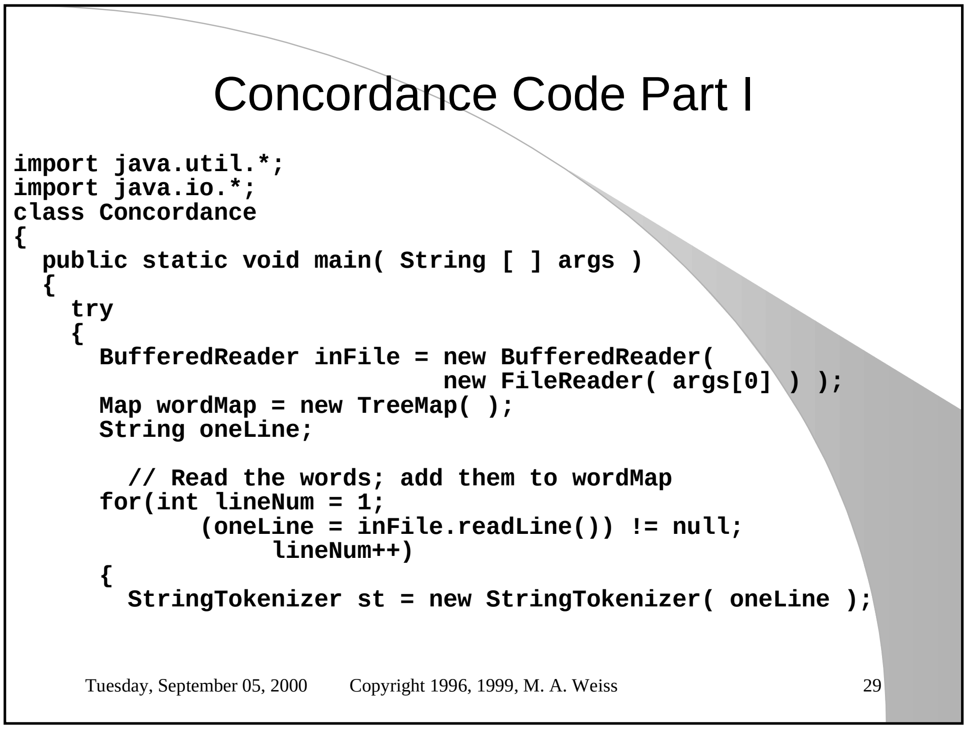
- **A collection of keys, values, or key/value pairs can be extracted from the map. An iterator can then traverse the collection.**

```
Set keySet( )
Collection values( )
Set entrySet( )
```

- **Each key/value entry is of the type `Map.Entry`. Use `getKey` and `getValue` on the `Map.Entry` object.**

Concordance Example

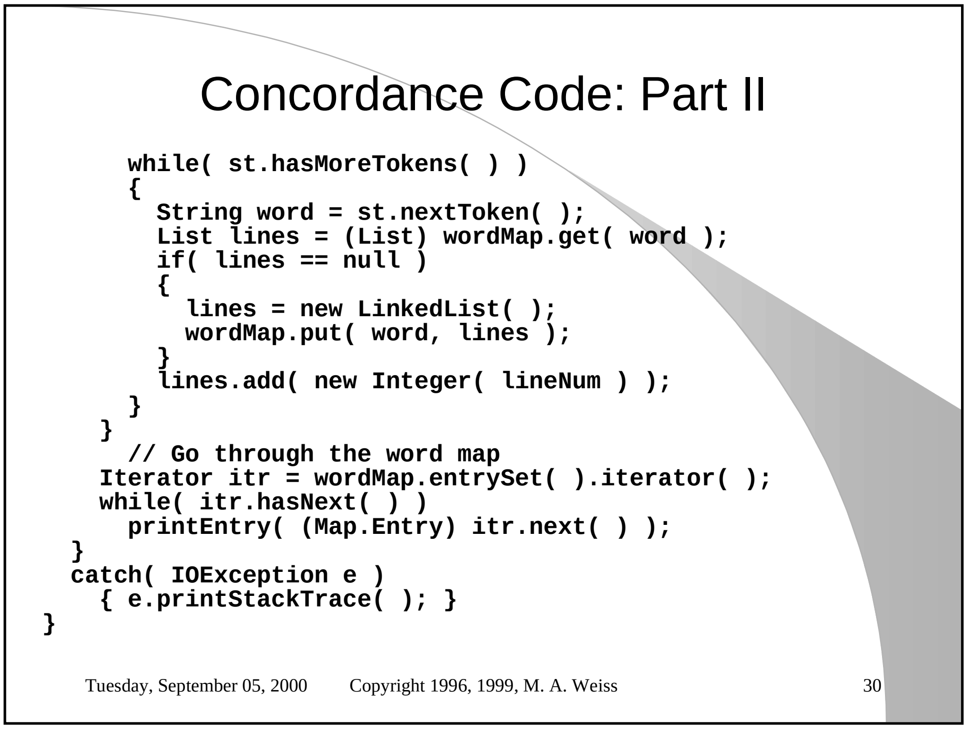
- **Read file containing words (several to a line).**
- **Output each unique word, and a list of line number on which it occurs.**
- **Basic algorithm: Use a `TreeMap`: map words to a linked list of lines. When the `TreeMap` is iterated, words come out in sorted order.**



Concordance Code Part I

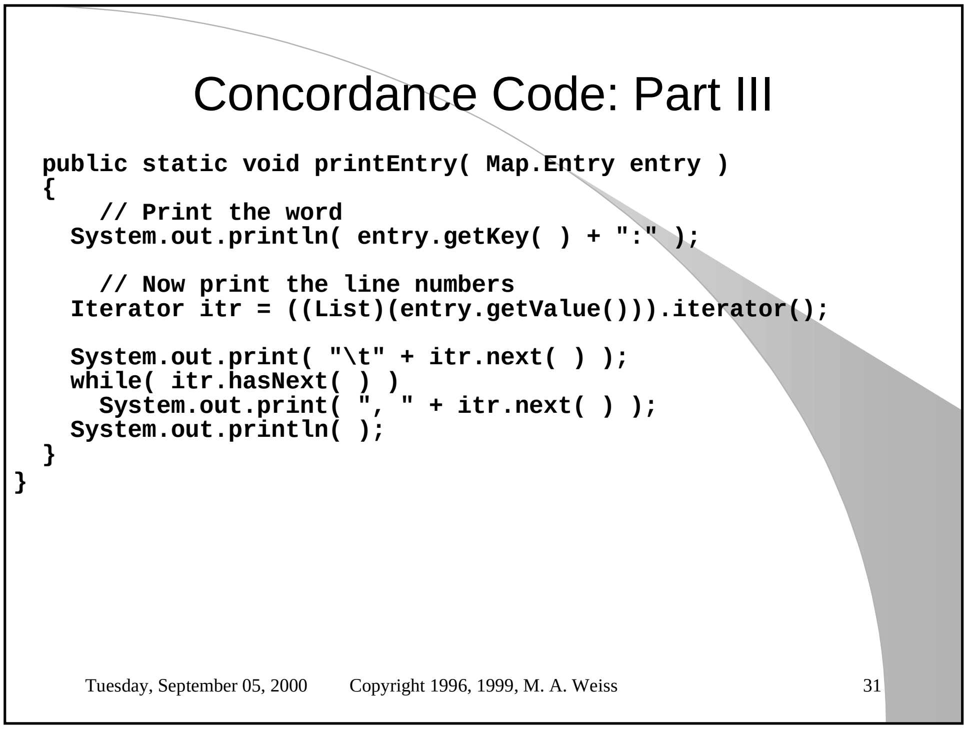
```
import java.util.*;
import java.io.*;
class Concordance
{
  public static void main( String [ ] args )
  {
    try
    {
      BufferedReader inFile = new BufferedReader(
                              new FileReader( args[0] ) );
      Map wordMap = new TreeMap( );
      String oneLine;

        // Read the words; add them to wordMap
      for(int lineNum = 1;
             (oneLine = inFile.readLine()) != null;
                  lineNum++)
      {
        StringTokenizer st = new StringTokenizer( oneLine );
```



Concordance Code: Part II

```
        while( st.hasMoreTokens( ) )
        {
          String word = st.nextToken( );
          List lines = (List) wordMap.get( word );
          if( lines == null )
          {
            lines = new LinkedList( );
            wordMap.put( word, lines );
          }
          lines.add( new Integer( lineNum ) );
        }
      }
        // Go through the word map
      Iterator itr = wordMap.entrySet( ).iterator( );
      while( itr.hasNext( ) )
        printEntry( (Map.Entry) itr.next( ) );
    }
    catch( IOException e )
      { e.printStackTrace( ); }
  }
```

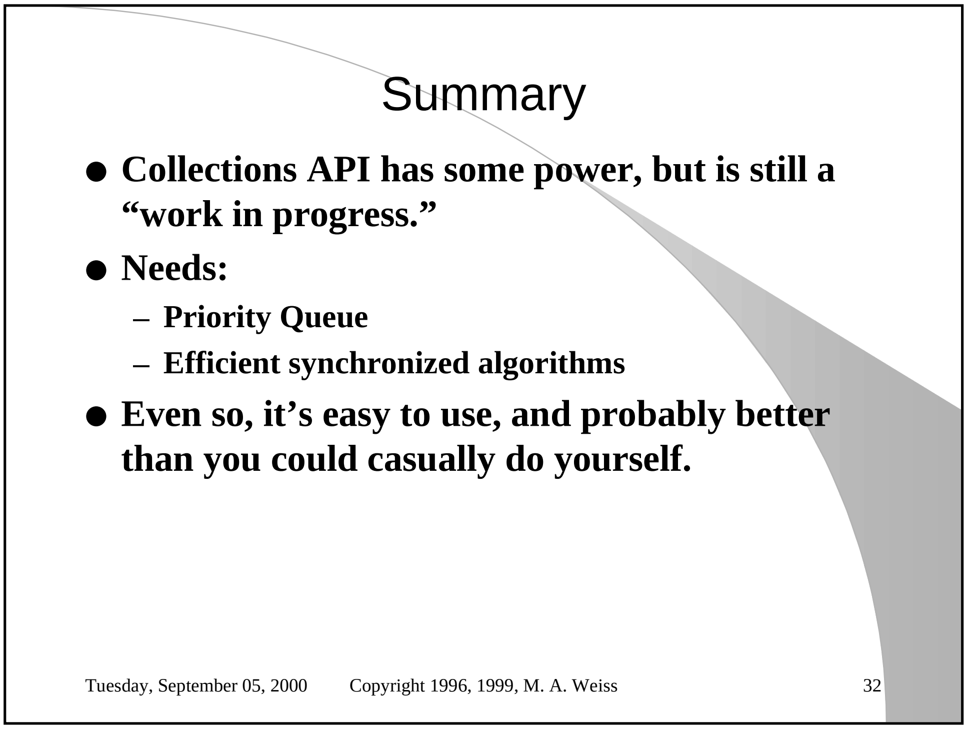


Concordance Code: Part III

```
  public static void printEntry( Map.Entry entry )
  {
      // Print the word
    System.out.println( entry.getKey( ) + ":" );

      // Now print the line numbers
    Iterator itr = ((List)(entry.getValue())).iterator();

    System.out.print( "\t" + itr.next( ) );
    while( itr.hasNext( ) )
      System.out.print( ", " + itr.next( ) );
    System.out.println( );
  }
}
```



Summary

- **Collections API has some power, but is still a “work in progress.”**
- **Needs:**
  - **Priority Queue**
  - **Efficient synchronized algorithms**
- **Even so, it’s easy to use, and probably better than you could casually do yourself.**