Collection Framework

"Code with Passion!"

Topics

- What is and Why Collections?
- Core Collection Interfaces
- Implementation classes
  - Set implementations
  - List implementations
  - Map implementations
- Natural ordering
- Iterator
- *Collections* utility class

What is and Why Collections?

What is a Collection?

- A “collection” object — sometimes called a “container” — is simply an object that contains other objects
  - You can think of it as a bag of objects
- Collections are used to store, retrieve, manipulate, and communicate aggregate data
- Typically, a collection represent real-life data items that form a group such as
  - Poker hand as a collection of cards
  - Mail folder as a collection of letters
  - Telephone directory as a map of names and phone numbers

What is Java Collection Framework?

- Java Collection framework provides a unified architecture for representing and manipulating collections
- It is a set of Java interfaces and implementation classes

Benefits of Java Collection Framework

- Reduces programming effort – you don't have to create your own scheme of representing and manipulating collections
  - Implementation is already provided in the JDK
- Increases program speed and quality
  - JDK implementation is highly optimized
- Allows interoperability among unrelated APIs
  - Collection interfaces are the common types by which APIs pass collection objects back and forth
- Fosters software reuse
  - New data structures that conform to the standard collection interfaces are by nature reusable

Java Interfaces & Implementations in Collection Framework

Java Collection Interfaces

- Java collection interfaces allow collections to be manipulated independently of the implementation details of their representation
  - Polymorphic behavior

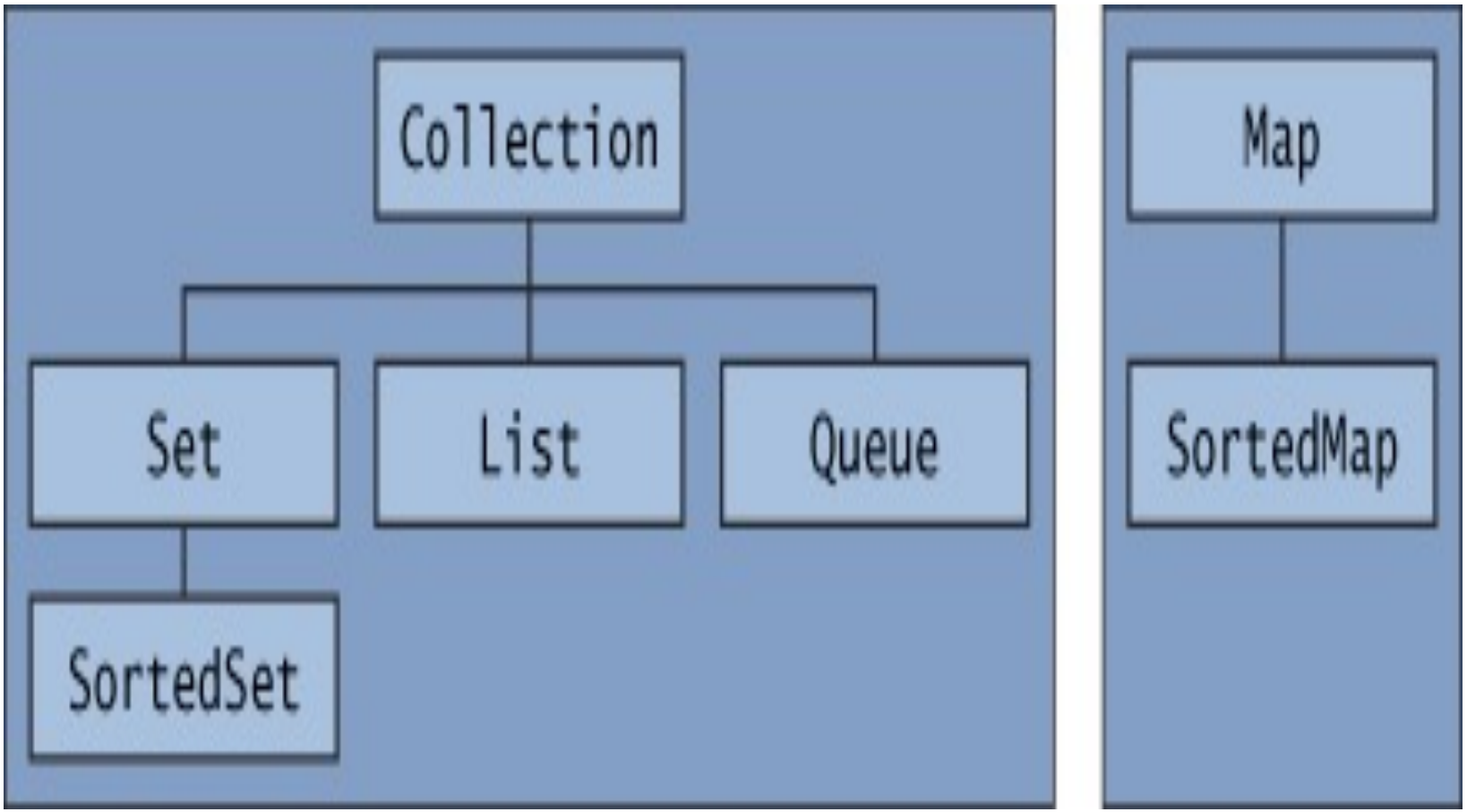
Java Collection Interfaces & Implementations

General-purpose Implementations

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Core Collection Interfaces

Core Collection Interfaces Hierarchy



Core Collection Interfaces

- Core collection interfaces are the foundation of the Java Collections Framework

- Core collection interfaces form an inheritance hierarchy among themselves
  - You can create a new custom Collection interface from them if you want to – but it is highly likely you don't have to

“Collection” Java Interface

“Collection” Java Interface

- The root of the collection hierarchy
  - Every collection object is a type of *Collection* interface
- Is used to pass collection objects around and to manipulate them when maximum generality is desired
  - Use *Collection* interface as a type
- JDK doesn't provide any direct implementations of this interface but provides implementations of more specific sub-interfaces, such as *Set* and *List*

“Collection” Interface

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Example: Usage “Collection” Interface as a Type

```
// Create a ArrayList collection object instance and assign it
// to Collection interface type.
Collection c1 = new ArrayList();

// Use methods of Collection interface.
//
// Polymorphic behavior is expected. For example,
// the add() implementation of ArrayList class will be
// invoked in the example below. And depending on the ,
// implementation, duplication could be allowed or not
// allowed.

boolean b2 = c1.add(new Integer(1));
boolean b1 = c1.isEmpty();
```

“Collection” Interface: add() and remove() Operations

add() and remove() methods of "Collection" Interface

- *add(E element)* method
  - Adds the specified element to the collection
  - *add()* method of *Set* interface follows "no duplicate" rule
  - Returns *true* if this collection changed as a result of the call

- *remove(Object element)* method
  - Removes the specified element from this collection, if it is present
  - Returns *true* if an element was removed as a result of this call

“Collection” Interface: Traversing

Two Schemes of Traversing a Collection Object

- for-each
  - The for-each construct allows you to traverse a collection using a for loop

```
for (Object o: collectionObject){
    // Do whatever you need to do with a member
    // object.
}
```

- Iterator
  - An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired

Iterator Interface

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

- *hasNext()* method
  - returns true if the iteration has more elements, (In other words, returns true if *next()* would return an element rather than throwing an exception.)
- *next()* method
  - returns the next element in the iteration
- *remove()* method
  - removes from the underlying collection the last element returned by the iterator
  - Can be called only once per call to *next()*

Example: Iterator

```
Iterator itr = aList.iterator();

// Remove 2 from ArrayList using Iterator's remove method.
String strElement = "";
while (itr.hasNext()) {

    // Iterator's next method returns an Object so we need to cast it into
    // appropriate class before using it.
    strElement = (String) itr.next();
    if (strElement.equals("2")) {
        itr.remove();
        break;
    }

}
```

“Collection” Interface: Bulk Operations

Bulk Operations

- *containsAll(Collection c)* — returns true if the target Collection object contains all of the elements in the specified Collection c.
  - Example: *boolean b = targetCollectionObject.containsAll(c);*
- *addAll(Collection c)* — adds all of the elements in the specified Collection c to the target Collection.
- *removeAll(Collection c)* — removes from the target Collection all of its elements that are also contained in the specified Collection c.
- *retainAll(Collection c)* — retains only those elements in the target Collection that are also contained in the specified Collection c.
- *clear()* — removes all elements from the target Collection.

“Collection” Interface: Array Operations

Array Operations

- The *Collection* interface as *toArray()* method, which is provided as a bridge between collections and older APIs that expect arrays on input
  - The array operations allow the contents of a Collection to be translated into an array

Note: The *Array* class also has *toList()* method, which converts an array into a List object

toArray() and toList() methods

```
// Convert a collection object into an array
// The length of array is the same as the number of elements in the collection
Object[] a = myCollection.toArray();


// Convert an array into a List
Person[] personArray =
            { new Person("Jong", 5), new Person("Jon", 12),
              new Person("Jon", 17), new Person("Mary", 13) };
List<Person> people = Arrays.asList(personArray);
```

Set Interface & Implementations

“Set” Interface

- Represents a collection that cannot contain duplicate elements
  - Cards comprising a poker hand
  - Courses making up a student's schedule
  - Processes running on a machine

- The implementation of the Set interface adds the restriction that duplicate elements are prohibited

“equals” operation of “Set” Interface

- *Set* implementations also have a stronger contract on the behavior of the *equals* and *hashCode* operations, allowing *Set* instances to be compared meaningfully even if their implementation types differ
  - Two Set instances are equal if they contain the same elements

Implementations of “Set” Interface

- *HashSet* class
- *TreeSet* class
- *LinkedHashSet* class

General-purpose Implementations

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

HashSet class

- *HashSet* is much faster than *TreeSet* but offers no ordering guarantees
- Offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets
- Mostly commonly used Set implementation

Caveats of Using HashSet

- Choosing an initial capacity that's too high can waste both space and time
- Choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity

TreeSet class

- Implements *SortedSet* interface
- Use it you need to use the operations in the *SortedSet* interface, or if value-ordered iteration is required

Example: Set Interface & TreeSet

```
public static void main(String[] args) {

    Set ts = new TreeSet();

    ts.add("one");
    ts.add("two");
    ts.add("three");
    ts.add("four");
    ts.add("three");

    // Words will be printed in the alphabetical order
    System.out.println("Members from TreeSet  in alphabetical order = " + ts);
}
```

Result:

Members from TreeSet in alphabetical order = [four, one, three, two]

LinkedHashSet

- Implemented as a hash table with a linked list running through it
- Provides insertion-ordered iteration (least recently inserted to most recently) and runs nearly as fast as HashSet
- Spares its clients from the unspecified, generally chaotic ordering provided by *HashSet* without incurring the increased cost associated with *TreeSet*

Example: Set Interface & LinkedHashSet

```
public static void main(String[] args) {

    Set ts2 = new LinkedHashSet();

    ts2.add(2);
    ts2.add(1);
    ts2.add(3);
    ts2.add(3);

    System.out.println("Members from LinkedHashSet in inserted order = " + ts2);

}
```

Result:

Members from LinkedHashSet in inserted order = [2, 1, 3]

Example: Set Implementation

```
public class Main {

    public static void main(String[] args) {

        Set s = new HashSet();      // Order is not guaranteed
        MyOwnUtilityClass.checkDuplicate(s, args);

        s = new TreeSet();          // Order according to values
        MyOwnUtilityClass.checkDuplicate(s, args);

        s = new LinkedHashSet(); // Order according to insertion
        MyOwnUtilityClass.checkDuplicate(s, args);
    }
}
```

Example: Set Implementation (Cont)

```
public class MyOwnUtilityClass {

    // Note that the first parameter type is set to
    // Set interface not a particular implementation
    // class such as HashSet.  This makes the caller of
    // this method to pass instances of different
    // implementations of Set interface while
    // this function picks up polymorphic behavior
    // depending on the actual implementation type
    // of the object instance passed.

    public static void checkDuplicate(Set s, String[] args){
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);

        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

Example: Set Implementations

Suppose the followings numbers are added in the following order

2

3

4

1

2

Set type = java.util.HashSet [3, 2, 4, 1] // order not guaranteed

Set type = java.util.TreeSet [1, 2, 3, 4] // order based on value

Set type = java.util.LinkedHashSet [2, 3, 4, 1] // order based on insertion

Synchronized Access to Set Object

- HashSet, TreeSet, and LinkedHashSet are not synchronized.
- If multiple threads access a set concurrently, and at least one of the threads modifies the set, it must be synchronized
  - This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the *Collections.synchronizedSet* method. This is best done at creation time, to prevent accidental unsynchronized access to the Set instance:
  - *Set s = Collections.synchronizedSet(new HashSet(...));*
  - *SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));*
  - *Set s = Collections.synchronizedSet(new LinkedHashSet(...));*

Lab:

Exercise 1: “Set” Collection Objects
1016_javase_collections.zip

Natural Ordering

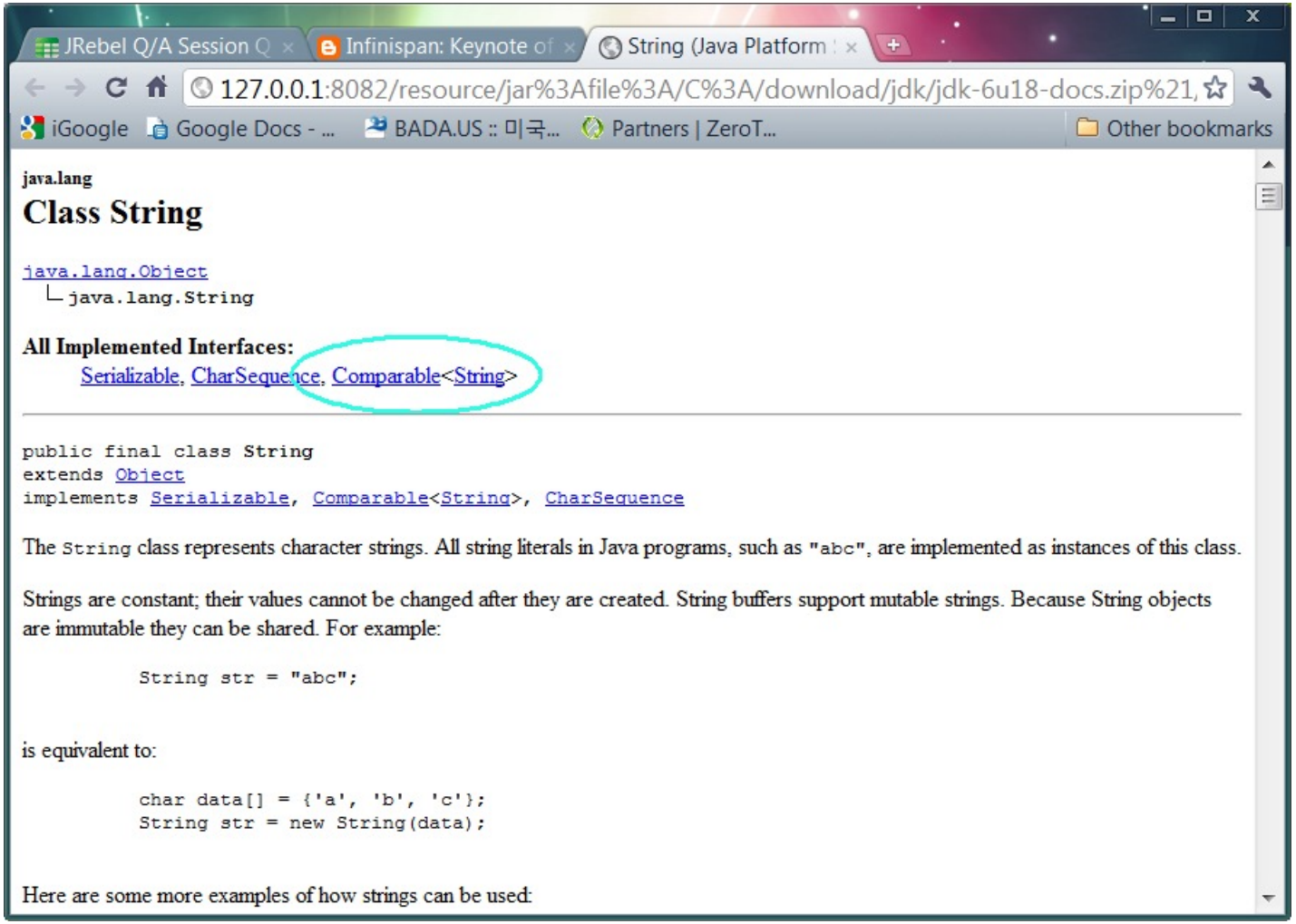
“SortedSet” Interface

- A Set that maintains its elements in order
  - Order is defined through natural ordering by default or through custom Comparator
- Several additional operations are provided to take advantage of the ordering
  - *comparator()* returns the comparator associated with this sorted set, or null if it uses its elements' natural ordering.
  - *first()* returns the first (lowest) element currently in this sorted set
  - *last()* returns the last (highest) element currently in this sorted set.

What is Natural Ordering?

- A class is said to have a natural ordering if it has implemented *java.lang.Comparable* interface
  - The *compareTo* method of the *java.lang.Comparable* interface is referred to as its natural comparison method
- Example classes that has natural ordering
  - String, Date, Integer, ...
- Example natural ordering
  - For String elements, natural ordering is alphabetical order
  - For Date elements, natural ordering is chronological order
  - For Integer elements, natural ordering is signed numerical order

Natural Ordering in String Class



List Interface & Implementations

“List” Interface

- An ordered collection (sometimes called a sequence)
- Lists can contain duplicate elements (as opposed to Set which does not allow duplicate elements)
- The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position)

Additional Operations Supported by "List" Interface over "Collection"

- Positional access — manipulates elements based on their numerical position in the list
- Search — searches for a specified object in the list and returns its numerical position
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature
- Range-view — performs arbitrary range operations on the list.

“List” Interface

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);
    boolean add(E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index,  Collection<? extends E> c);

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o); // Returns the index in this list of the last occurrence of the
                               // specified element, or -1 if this list does not contain this element.

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

Implementations of “List” Interface

General-purpose Implementations

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

ArrayList

- Resizable-array implementation of the List interface
- Offers constant-time positional access
- Each ArrayList instance has a capacity
  - The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size.
  - As elements are added to an ArrayList, its capacity grows automatically.
  - An application can increase the capacity of an ArrayList instance before adding a large number of elements using the *ensureCapacity* operation. This may reduce the amount of incremental reallocation.

LinkedList

- Linked list implementation of the List interface
- Offers constant-time insertions or removals
- Not a good choice for random access

Synchronized Access to List Object

- Neither ArrayList nor LinkedList is not synchronized.
- If multiple threads access a set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
  - If multiple threads access a list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the *Collections.synchronizedList* method. This is best done at creation time, to prevent accidental unsynchronized access to the list:
  - *List list = Collections.synchronizedList(new ArrayList(...));*
  - *List list = Collections.synchronizedList(new LinkedList(...));*

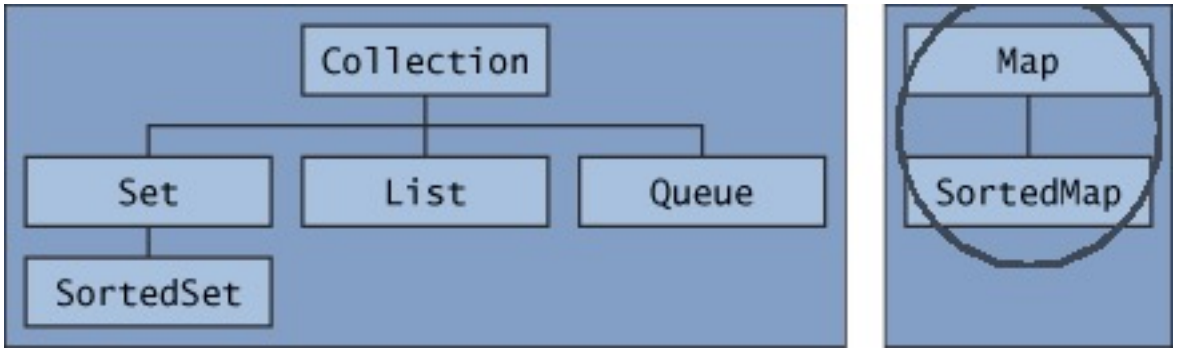
Lab:

Exercise 2: “List” Collection Objects
1016_javase_collections.zip

Map Interface & Implementations

“Map” Interface

- Handles key/value pairs
- A Map cannot contain duplicate keys



“Map” Interface (Java SE 5)

```
public interface Map<K,V> {
   // Basic operations
   V put(K key, V value);
   V get(Object key);
   V remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   // Bulk operations
   void putAll(Map<? extends K, ? extends V> m);
   void clear();

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

   // Interface for entrySet elements
   public interface Entry {
      K getKey();
      V getValue();
      V setValue(V value);
   }
}
```

“SortedMap” Interface

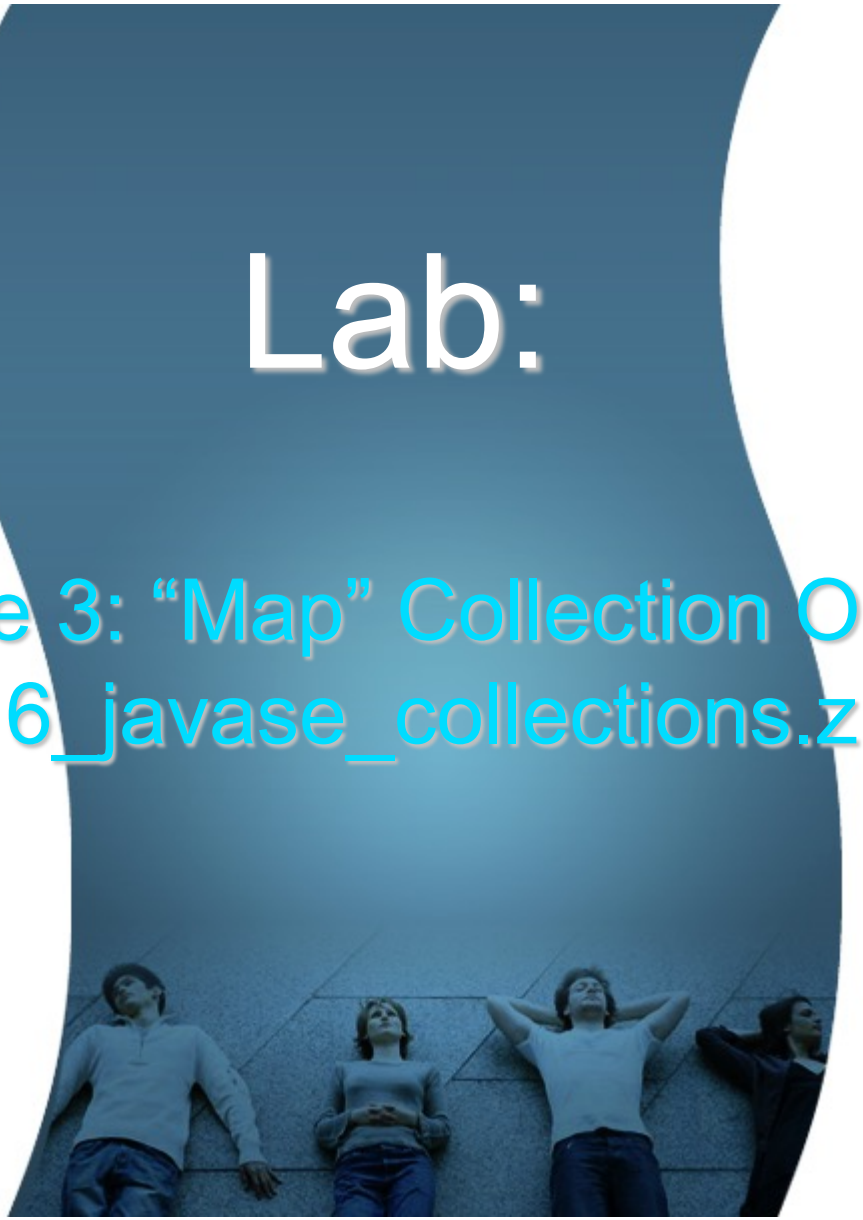
- A Map that maintains its mappings in ascending key order
  - This is the Map analog of SortedSet
- Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories

Implementations of “Map” Interface

- HashMap
  - Use it you want maximum speed and don't care about iteration order
  - Most commonly used Map implementation
- TreeMap
  - Implements SortedMap interface
  - Use it when you need key-ordered Collection-view iteration
- LinkedHashMap
  - Use if you want near-HashMap performance and insertion-order iteration

Lab:

Exercise 3: “Map” Collection Objects
1016_javase_collections.zip



Iterator

What is and Why Iterator?

- Often, you will want to cycle through the elements in a collection
  - For example, you might want to display each element
- The easiest way to do this is to employ an iterator, which is an object that implements either the *Iterator* or the *ListIterator* interface
  - *Iterator* enables you to cycle through a collection, obtaining or removing elements
  - *ListIterator* extends Iterator to allow bidirectional traversal of a list, and the modification of element

How to Use Iterator?

- Each of the collection classes provides an *iterator( )* method that returns an iterator to the start of the collection
  - For collections that implement List, you can also obtain an iterator by calling *ListIterator*

- In general, to use an iterator to cycle through the contents of a collection, follow these steps:
  - Obtain an iterator to the start of the collection by calling the collection's iterator( ) method
  - Set up a loop that makes a call to *hasNext( ).* Have the loop iterate as long as *hasNext( )* returns true
  - Within the loop, obtain each element by calling next( )

Lab:

Exercise 4: Iterator
1016_javase_collections.zip

Collections Utility Class

"*Collections*" Utility Class

- Provides methods to return an instance of an empty Set, List, and Map — emptySet, emptyList, and emptyMap
  - Example: *tourist.declarePurchases(Collections.emptySet());*
- Provides methods for sorting and shuffling
  - *Collections.sort(l);*
  - *Collections.shuffle(l);*

Utility Functions (Algorithms) Supported In "*Collections*" Class

- Sorting
- Shuffling
- Routine data manipulation
- Searching
- Composition
- Find extreme values

Sorting

- The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship

- Two forms of the operations
  - takes a List and sorts it according to its elements' natural ordering
  - takes a Comparator in addition to a List and sorts the elements with the Comparator

Natural Ordering Again

- The type of the elements in a List already implements *Comparable* interface
- Examples
  - If the List consists of String elements, it will be sorted into alphabetical order
  - If it consists of Date elements, it will be sorted into chronological order
  - String and Date both implement the Comparable interface. Comparable implementations provide a natural ordering for a class, which allows objects of that class to be sorted automatically
- If you try to sort a list, the elements of which do not implement Comparable, Collections.sort(list) will throw a ClassCastException.

Example: Sorting by Natural Order of List

```
// Set up test data
 String n[] = {
     new String("John"),
     new String("Karl"),
     new String("Groucho"),
     new String("Oscar")
 };

 // Create a List from an array
 List l = Arrays.asList(n);

 // Perform the sorting operation
 Collections.sort(l);
```

Sorting by Comparator Interface

- Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order
- Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering

Example: Sorting by Comparator Interface

```
// Set up test data
 ArrayList u2 = new ArrayList();
 u2.add("Beautiful Day");
 u2.add("Stuck In A Moment You Can't Get Out Of");
 u2.add("Elevation");
 u2.add("Walk On");
 u2.add("Kite");
 u2.add("In A Little While");
 u2.add("Wild Honey");
 u2.add("Peace On Earth");
 u2.add("When I Look At The World");
 u2.add("New York");
 u2.add("Grace");

 Comparator comp = MyComparators.stringComparator();
 Collections.sort(u2, comp);
 System.out.println(u2);
```

Shuffling

- The shuffle algorithm does the opposite of what sort does, destroying any trace of order that may have been present in a List
- Useful in implementing games of chance
  - could be used to shuffle a List of Card objects representing a deck
  - generating test cases

Routine Data Manipulation

- The *Collections* class provides five algorithms for doing routine data manipulation on List objects
  - *reverse* — reverses the order of the elements in a List.
  - *fill* — overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.
  - *copy* — takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
  - *swap* — swaps the elements at the specified positions in a List.
  - *addAll* — adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

Searching

- The Collections class has *binarySearch()* method for searching a specified element in a sorted List

```
// Set up testing data
String name[] = {
        new String("Sang"),
        new String("Shin"),
        new String("Boston"),
        new String("Passion"),
        new String("Shin"),
};

List l = Arrays.asList(name);

int position = Collections.binarySearch(l, "Boston");
System.out.println("Position of the searched item = " + position);
```

Misc. Functions

- Collections.frequency(myCollection, myElement)
  - Returns the number of elements in the specified collection equal to the specified object
- Collections.disjoint(myCollection1, myCollection2)
  - Returns true if the two specified collections have no elements in common.

Lab:

Exercise 5: Utility Classes
1016_javase_collections.zip

Code with Passion!