

Java Collection Framework

Sean P. Strout (sps@cs.rit.edu)

Robert Duncan(rwd@cs.rit.edu)

1/17/2006

Java Collection Framework

1



Java Collection Framework

- In computer science, a **data structure** is a way of storing data in a computer such that it can be used efficiently
- So far the only data structure you've learned to use is the array
- Java has a complete framework for collection classes which we will be exploring next

1/17/2006

Java Collection Framework

2

- The Java collections framework is made up of a set of interfaces and classes for working with groups of objects
- The Java Collections Framework provides
 - **Interfaces:** abstract data types representing collections.
 - **Implementations:** concrete implementations of the collection interfaces.
 - **Algorithms:** methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces.

- In math, a **set** is a collection of unique elements

$$X = \{ 1, 3, 7, 9 \}$$

$$Y = \{ 1, 2, 3, 4, 5 \}$$

$$Z = \{ 1, 9 \}$$

- What are the results of the following operations?

$$1. \quad 3 \in X$$


$$2. \quad 4 \notin X$$

$$3. \quad X \cup Y$$

$$4. \quad X \cap Y$$

$$5. \quad Z \subset X$$

$$6. \quad X \supset Z$$



Department of
Computer Science

Collection Interface

- The root interface for manipulating a collection of objects is `Collection`

java.util.Collection


- + add(o : Object) : boolean
- + addAll(c : Collection) : boolean
- + clear() : void
- + contains(o : Object) : boolean
- + containsAll(c : Collection) : boolean
- + equals(o : Object) : boolean
- + hashCode() : int
- + isEmpty() : boolean
- + iterator() : Iterator
- + remove(o : Object) : boolean
- + removeAll(c : Collection) : boolean
- + retainAll(c : Collection) : boolean
- + size() : int
- + toArray() : Object []
- + toArray(array : Object[]) : Object[]

→

java.util.Iterator

- + hasNext() : boolean
- + next() : Object
- + remove() : void

1/17/2006
Java Collection Framework
5



Department of
Computer Science

Collection Interface

- Consider our previous set example:

$$X = \{ 1, 3, 7, 9 \}$$

- What `Collection` method would we use to insert these four elements into the set?
 - And how can we tell if it was added or not?
- What method would we use to check whether the set has the number 7?
- How can we tell how large the set is?
- How can we tell what elements are in the set?

1/17/2006
Java Collection Framework
6

- In order to **traverse** a collection you must use an Iterator
 - Collection objects have an `iterator` method which returns an iterator for itself
- An **iterator** is a special object which gives us sequential access to the objects in a collection

$$x = \{ 1, 3, 7, 9 \}$$

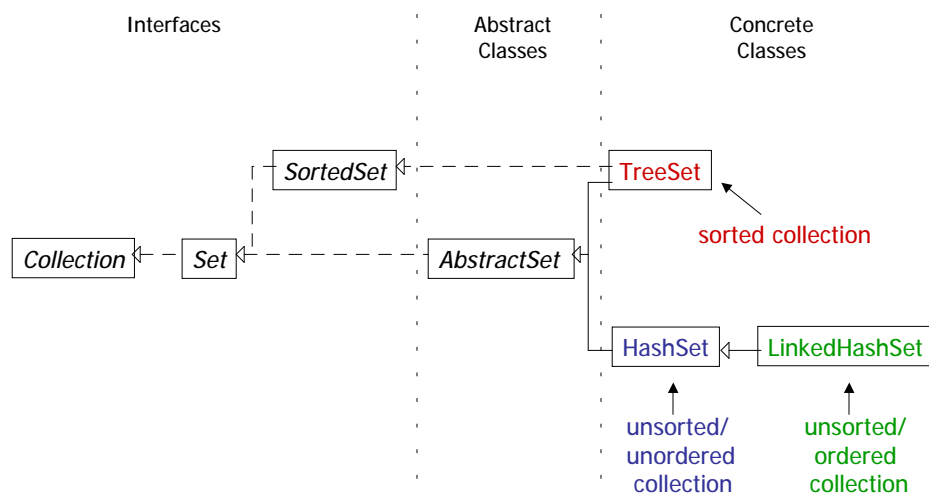
```
// X is a set which implements Collection

Iterator iter = x.iterator();
System.out.println(iter.next()); // prints out 1
System.out.println(iter.next()); // prints out 3
iter.hasNext();                  // true
```

1/17/2006

Java Collection Framework

7



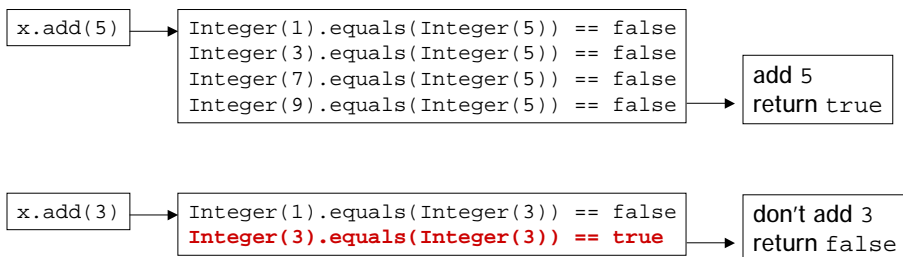
- Java has several classes which allow us to mimic the behavior of a set

1/17/2006

Java Collection Framework

8

- All sets have the restriction that **duplicate elements may not be added**
- In order to enforce this, the set classes must compare the object being added to all the elements in the set

$$x = \{ 1, 3, 7, 9 \}$$


1/17/2006

Java Collection Framework

9

- A **hash code** is an integer value which uniquely identifies an object
- It is computable using a simple formula based on the type of object, using the `hashCode` method
- Most classes in the Java API implement their own `hashCode` method
- Internally the data is stored in a **hash table**

1/17/2006

Java Collection Framework

10



HashCode.java

```
public class HashCode {
    public static void main (String args[]) {
        Integer myInt = 10;    // auto-box (Java 1.5+)    Double myDouble = 1.79;
        Character myChar = 'a';    String myString = new String ( "abc" );

        // Hashcode for an Integer is its integer value
        // i.e. 10 = 10
        System.out.println("Integer = " + myInt + ", hashCode = " + myInt.hashCode());

        // Hashcode for a Double is: (int)(v ^ (v >> 32))
        // 1.79 = 901895027, (int)(4610740262505640100 ^ 1073521623)
        System.out.println("Double = " + myDouble + ", hashCode = " + myDouble.hashCode());

        // Hashcode for a Character is the unicode value
        // i.e.: 'a' = 97
        System.out.println("Character = " + myChar + ", hashCode = " + myChar.hashCode());

        // Hashcode for a String is: s0 * 31^(n-1) + s1 * 31^(n-2) + ... + sn-1
        // i.e.: "abc" = 97 * 31^2 + 98 * 31 + 99 = 96354
        System.out.println("String = " + myString + ", hashCode = " + myString.hashCode());
    } // main
} // HashCode
```

1/17/2006

Java Collection Framework

11



HashCode Output

OUTPUT:

```
Integer = 10, hashCode = 10
Double = 1.79, hashCode = 901895027
Character = a, hashCode = 97
String = abc, hashCode = 96354
```

1/17/2006

Java Collection Framework

12

- To create a HashSet of integers:

```
Set set = new HashSet(); // old way
Set<Integer> set = new HashSet<Integer>(); // new way
```

- For efficiency sake, a HashSet does not necessarily store the elements in the order they were inserted

```
public class TestHashSet {
    public static void main (String args[]) {
        String phrase = "It was the best of times, it was the worst of times."; // The phrase
        Set<String> set = new HashSet<String>(); // Create the hash set

        // Extract words from the phrase using a StringTokenizer.
        // The characters that separate words in this phrase are
        // space, comma and period.
        StringTokenizer st = new StringTokenizer(phrase, " ,.");

        // Put each word into the set
        while (st.hasMoreTokens()) {
            set.add(st.nextToken());
        }
        // Print out the whole set
        System.out.println("The set: " + set);

        // Display the elements in the set using an Iterator
        System.out.print("The elements:");
        Iterator iter = set.iterator();
        while (iter.hasNext()) {
            System.out.print(" " + iter.next());
        }
    } // main
} // TestHashSet
```

OUTPUT:

The set: [the, of, it, It, times, best, worst, was]

The elements: the of it It times best worst was

- A `LinkedHashSet` is a subclass of `HashSet`
- It supports ordering of the elements in the set (ordering is not the same as sorting)
- The elements can be retrieved in the order in which they were inserted into the set
- Internally the elements are stored in a **linked list**



TestLinkedHashSet.java

```

public class TestLinkedHashSet {
    public static void main (String args[]) {
        String phrase = "It was the best of times, it was the worst of times."; // The phrase

        // Create a linked hash set
        Set<String> set = new LinkedHashSet<String>();

        // Extract words from the phrase using a StringTokenizer.
        // The characters that separate words in this phrase are
        // space, comma and period.
        StringTokenizer st = new StringTokenizer(phrase, " ,.");

        // Put each word into the set
        while (st.hasMoreTokens()) {
            set.add(st.nextToken());
        }

        // Print out the whole set
        System.out.println("The set: " + set);

        // Display the elements in the set using an Iterator
        System.out.print("The elements:");
        Iterator iter = set.iterator();
        while (iter.hasNext()) {
            System.out.print(" " + iter.next());
        }
    } // main
} // TestLinkedHashSet

```

1/17/2006

Java Collection Framework

17



TestLinkedHashSet Output

OUTPUT:

The set: [It, was, the, best, of, times, it, worst]

The elements: It was the best of times it worst

1/17/2006

Java Collection Framework

18

- Java 1.5 has simplified iteration over a collection:

```

Iterator iterator = set.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next() + " ");
}

```

The old way is to get an iterator and continually call `hasNext` and `next`

```

for (String element: set) {
    System.out.println(element + " ");
}

```

The new way is to use a `for` statement which wraps an iterator internally

- The previous sets do not sort the elements in relation to each other

1 5 3 0 9 4 2

This is the order of the elements when iterated over:

HashSet: 2 4 9 1 3 5 0

LinkedHashSet: 1 5 3 0 9 4 2

- A `TreeSet` implements the `SortedSet` interface which means all elements are compared when inserted

TreeSet: 0 1 2 3 4 5 9

- Tree's are implemented using a special form of a **binary search tree** known as a Red/Black Tree
 - <http://www.eecs.uc.edu/~franco/C321/html/RedBlack/redblack.html>

```

public class TestTreeSet {
    public static void main (String args[]) {
        String phrase = "It was the best of times, it was the worst of times."; // The phrase

        // Create a linked hash set
        Set<String> set = new TreeSet<String>();

        // Extract words from the phrase using a StringTokenizer.
        // The characters that separate words in this phrase are
        // space, comma and period.
        StringTokenizer st = new StringTokenizer(phrase, " ,.");

        // Put each word into the set
        while (st.hasMoreTokens()) {
            set.add(st.nextToken());
        }

        // Print out the whole set
        System.out.println("The set: " + set);

        // Display the elements in the set using an Iterator
        System.out.print("The elements:");
        Iterator iter = set.iterator();
        while (iter.hasNext()) {
            System.out.print(" " + iter.next());
        }
    } // main
} // TestTreeSet

```

OUTPUT:

The set: [It, best, it, of, the, times, was, worst]

The elements: It best it of the times was worst

- All elements added to the tree must be comparable in one of two ways:
 - **Natural order comparison** means the objects implements the `compareTo` method in the `Comparable` interface
 - i.e. the `String` and wrapper classes use this
 - **Order by comparator** means the objects implement the `compare` method in the `Comparator` interface

- Sometimes you want to put different types of elements into a tree set which do not implement `Comparable`
- You can make a custom class which implements the `Comparator` interface

<code>java.util.Comparator</code>
<code>+ compare(Object element1, Object element2) : int</code> <code>+ equals(Object element) : boolean</code>

- With the `compare` method, you can control how potential heterogeneous objects will be compared

- Recall the abstract `Shape` example
- Suppose we want to store `Square`'s, `Circle`'s and `Triangle`'s in a `TreeSet`
- We want to sort the shapes by their area
- We didn't write those classes to implement `Comparable`
- The solution is to write a custom comparator class which allows the `TreeSet` to sort the shapes



TestTreeSetWithComparator.java

```

public class TestTreeSetWithComparator {
    public static void main(String args[]) {
        Set<Shape> shapeSet = // Create a tree set using a comparator
            new TreeSet<Shape>(new ShapeComparator());
        // Add a bunch of different shapes into the tree set
        Circle c1 = new Circle(5);
        System.out.println("\nADDING: " + c1);
        shapeSet.add(c1);
        Square s1 = new Square(4);
        System.out.println("\nADDING: " + s1);
        shapeSet.add(s1);
        Square s2 = new Square(10);
        System.out.println("\nADDING: " + s2);
        shapeSet.add(s2);
        Square s3 = new Square(4);
        System.out.println("\nADDING: " + s3);
        shapeSet.add(s3); // same as s1, should skip
        Triangle t1 = new Triangle(5,3);
        System.out.println("\nADDING: " + t1);
        shapeSet.add(t1);
        System.out.println("=====");
        // Loop over the elements in the tree set
        // using the new for loop in Java 1.5
        System.out.println("\nHere's the tree set:");
        for (Shape element: shapeSet) {
            System.out.println(element + ", area= " + element.getArea());
        }
    } // main
} // TestTreeSetWithComparator
1/17/2006

```

Java Collection Framework

27



TestTreeSetWithComparator Output

```

ADDING: [Circle] radius=5.0
In compare method of ShapeComparator.
Comparing:
s1=[Square] side=4.0
s2=[Circle] radius=5.0
s1 less than s2

ADDING: [Square] side=10.0
In compare method of ShapeComparator.
Comparing:
s1=[Square] side=10.0
s2=[Circle] radius=5.0
s1 greater than s2

ADDING: [Square] side=4.0
In compare method of ShapeComparator.
Comparing:
s1=[Square] side=4.0
s2=[Circle] radius=5.0
s1 less than s2

In compare method of ShapeComparator.
Comparing:
s1=[Triangle] base=5.0 and height=3.0
s2=[Square] side=4.0
s1 equals s2

ADDING: [Triangle] base=5.0 and height=3.0
In compare method of ShapeComparator.
Comparing:
s1=[Triangle] base=5.0 and height=3.0
s2=[Circle] radius=5.0
s1 less than s2

In compare method of ShapeComparator.
Comparing:
s1=[Triangle] base=5.0 and height=3.0
s2=[Square] side=4.0
s1 less than s2

=====

Here's the tree set:
[Triangle] base=5.0 and height=3.0, area= 3.75
[Square] side=4.0, area= 16.0
[Circle] radius=5.0, area= 78.53981633974483
[Square] side=10.0, area= 100.0

```

1/17/2006

Java Collection Framework

28

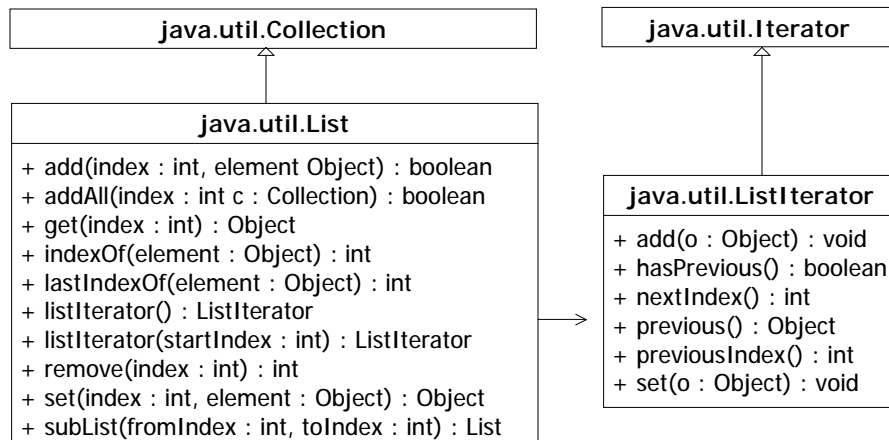
- To allow **duplicate** elements to be stored in a collection, you need to use a **list**
- A list allows users to store or remove elements at specific locations
- The user can access any element in a list by an index
- The user can determine the index of any element in the list
- The index is zero based, just like with arrays

1/17/2006

Java Collection Framework

29

- The `List` interface adds **position-oriented operations** and a new `ListIterator`



1/17/2006

Java Collection Framework

30

- Consider the following list of elements:

```
myList = 0 5 6 9 2 1 2 1
```

- Answer the following questions (independently):
 - How do we insert the element 8 after element 6?
 - How do we get the third element in the list, 6?
 - How do we find what location 9 is in the list?
 - How do we get the location of the last 1 in the list?
 - How do we remove the first 2 from the list?
 - How do you get the sublist containing elements: 6 9 2 ?
 - How do you change element 5 to 7?
 - What's the easiest way to make the list look like:
0 5 6 4 3 7 9 2 1 2 1

1/17/2006

Java Collection Framework

31

- Write a Java loop which prints out the elements from last to first

```
myList = 0 5 6 9 2 1 2 1
output: 1 2 1 2 9 6 5 0
```

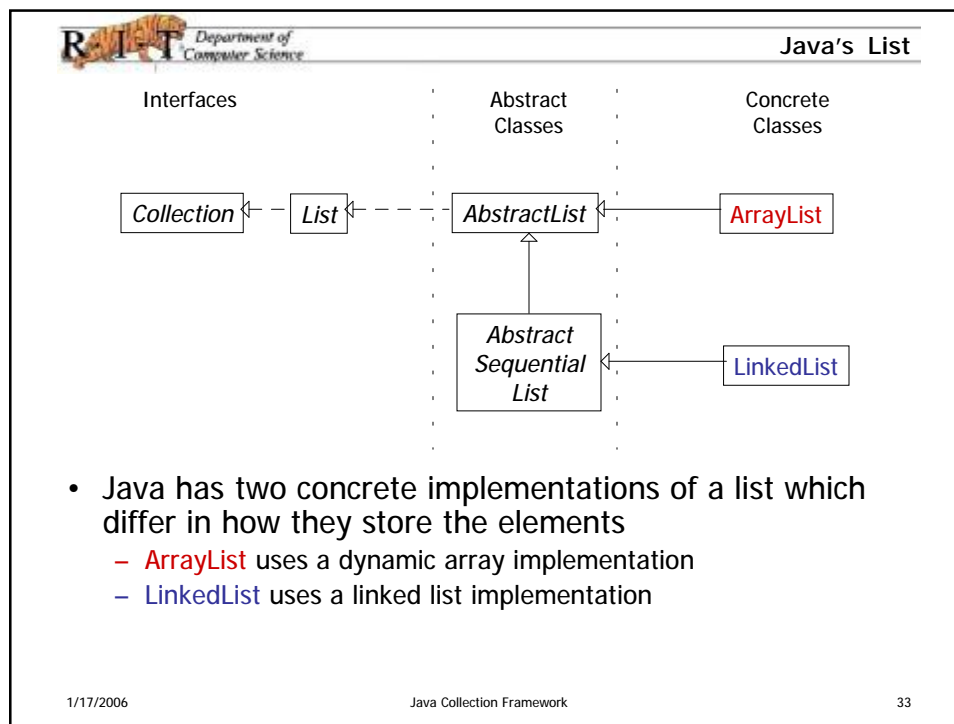
- Using an iterator, write the Java code which replaces all occurrences of 1 with 7

```
myList = 0 5 6 9 2 7 2 7
```

1/17/2006

Java Collection Framework

32



ArrayList vs LinkedList

- Consider the list: 0 1 3 5 9 7
- ArrayList:

0	1	2	3	4	5
0	1	3	5	9	7
- LinkedList:

0	↔	1	↔	3	↔	5	↔	9	↔	7
---	---	---	---	---	---	---	---	---	---	---
- Which collection is faster at **random access**? i.e. print out the elements in the list with odd numbered indices
- Which collection is faster if you require insertion or deletion of elements in the middle of the list? i.e. insert 4 between 3 and 5 and delete element 9


1/17/2006 Java Collection Framework 34

- An ArrayList uses a resizable-array implementation of the List implementation
- Each ArrayList has a **capacity** which is the size of the array used to store the elements
 - The size is always less than or equal to the capacity
 - If an ArrayList is constructed without an initial capacity, the default is 10 elements
- As elements are added, the capacity grows automatically

```

public class TestArrayList {
    final static int SIZE = 5;
    public static void main(String args[]) {
        // Create an array list with a particular size
        ArrayList<Integer> arrayList = new ArrayList<Integer>(SIZE);
        for (int i=10; i>0; i--) { // Add elements from 10 to 1
            arrayList.add(i);
        }
        System.out.println("Array list: " + arrayList);
        System.out.print("The elements in reverse order:");
        ListIterator iter = arrayList.listIterator(arrayList.size());
        while (iter.hasPrevious()) {
            System.out.print(" " + iter.previous());
        }
        System.out.println();
        System.out.println("Size: " + arrayList.size());
        System.out.println("Contains 5? " + arrayList.contains(5));
        System.out.println("Contains 20? " + arrayList.contains(20));
        System.out.println("Index of 2: " + arrayList.indexOf(2));
        arrayList.remove(3);
        System.out.println("Remove index 3: " + arrayList);
        ArrayList<Integer> newArrayList = new ArrayList<Integer>();
        newArrayList.add(20);
        newArrayList.add(21);
        newArrayList.add(22);
        arrayList.addAll(4, newArrayList);
        System.out.println("Adding new array list: " + newArrayList + " at index 4: " + arrayList);
    } // main
} // TestArrayList

```




Department of
Computer Science

TestArrayList Output

OUTPUT:

Array list: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
 The elements in reverse order: 1 2 3 4 5 6 7 8 9 10
 Size: 10
 Contains 5? true
 Contains 20? false
 Index of 2: 8
 Remove index 3: [10, 9, 8, 6, 5, 4, 3, 2, 1]
 Adding new array list: [20, 21, 22] at index 4: [10, 9, 8, 6, 20, 21, 22, 5, 4, 3, 2, 1]

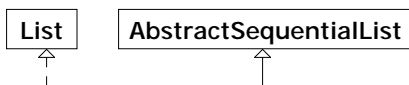
1/17/2006
Java Collection Framework
37



Department of
Computer Science

LinkedList

- `LinkedList` has utility methods for retrieving, inserting and removing elements from both ends of the list



```

classDiagram
    class List
    class AbstractSequentialList
    class java_util_LinkedList["java.util.LinkedList"]
    java_util_LinkedList --|> List
    java_util_LinkedList --|> AbstractSequentialList
  
```

java.util.LinkedList

- + LinkedList()
- + LinkedList(c : Collection)
- + addFirst(o : Object) : void
- + addLast(o : Object) : void
- + getFirst() : Object
- + getLast() : Object
- + removeFirst() : Object
- + removeLast() : Object

1/17/2006
Java Collection Framework
38



TestLinkedList.java

```

public class TestLinkedList {
    final static int SIZE = 5;
    public static void main(String args[]) {
        // Create a linked list
        LinkedList<Integer> linkedList = new LinkedList<Integer>();

        // Add elements from 10 to 1
        for (int i=10; i>0; i--) {
            linkedList.add(i);
        }

        System.out.println("Linked list: " + linkedList);
        System.out.println("getFirst: " + linkedList.getFirst());
        System.out.println("getLast: " + linkedList.getLast());
        linkedList.addFirst(11);
        System.out.println("addFirst 11: " + linkedList);
        linkedList.addLast(0);
        System.out.println("addLast 0: " + linkedList);
        linkedList.add(3, 100);
        System.out.println("add(3, 100): " + linkedList);

        System.out.print("The list in reverse order: ");
        ListIterator iter = linkedList.listIterator(
            linkedList.size());
        while (iter.hasPrevious()) {
            System.out.print(" " + iter.previous());
        }
    } // main
} // TestLinkedList
1/17/2006

```

Java Collection Framework

39



TestLinkedList Output

OUTPUT:

Linked list: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

getFirst: 10

getLast: 1

addFirst 11: [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

addLast 0: [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

add(3, 100): [11, 10, 9, 100, 8, 7, 6, 5, 4, 3, 2, 1, 0]

The list in reverse order: 0 1 2 3 4 5 6 7 8 100 9 10 11

1/17/2006

Java Collection Framework

40

- How would you describe a **dictionary** to an alien (who presumably speaks good English...)?
- How is a dictionary organized?
 - What are the key components?
 - How do you find what you are looking for in a dictionary?
- Does any information have to be unique?
- What collection that we have already seen do dictionaries most resemble?

- In computer science, a **map** is a collection which stores **keys** that "map to" corresponding **values**
- The key can be any kind of object (which is hashable), but it must be unique
 - The values, on the other hand, may be duplicated
- The key is used as an index to lookup one value
- Common map operations include
 - **Query**: Is a given key in the map?
 - **Update**: Change the value for a given key
 - **Obtain a set of keys**
 - **Obtain a collection of values**

- Consider a program which **counts the occurrences of words** in the following phrase:

"It was the best of times, it was the worst of times"

- What should we store for keys and values?
- Draw a picture which shows the contents of the map
 - What would the (key, value) pairs in the map be?
- What are the set of keys in the map?
- What are the collection of values in the map?

1/17/2006

Java Collection Framework

43

java.util.Map

```
+ clear() : void
+ containsKey(key : Object) : boolean
+ containsValue(value : Object) : boolean
+ entrySet() : Set
+ get(key : Object) : Object
+ isEmpty() : boolean
+ keySet() : Set
+ put(key : Object, value : Object) : Object
+ putAll(m : Map) : void
+ remove(key : Object) : Object
+ size() : int
+ values() : Collection
```

- The `Map` interface provides methods for querying, updating, and obtaining a collection of values and a set of keys

1/17/2006

Java Collection Framework

44

- Consider the following map of courses being taught and the number of students enrolled in each course:

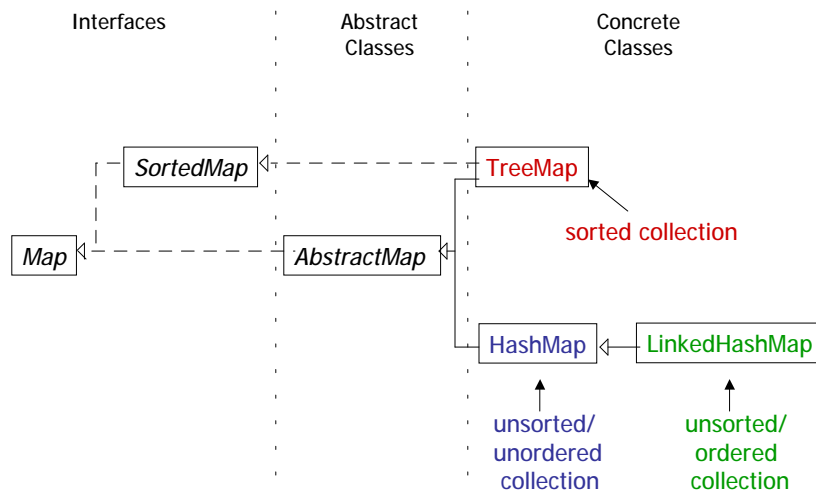
```
myMap = { "cs2"=45, "cs2s"=15, "cs4"=49 }
```

- Answer the following questions:
 - How do you check if I'm teaching the course "cs2" ?
 - How do you check if any course I'm teaching has 15 students ?
 - How do you find how many students are in "cs2" ?
 - How do you add another course to the map: "cg1"=30 ?
 - How do you get the complete list of courses I am teaching?
 - How do you find the total number of students I am teaching?

1/17/2006

Java Collection Framework

45



- The map classes follow the same format as sets

1/17/2006

Java Collection Framework

46

- With Java 1.5, you can create a map with specific keys and values using generics


key value

```
HashMap<String, Integer> myMap =  
    new HashMap<String, Integer>();
```

- Use `HashMap` when you don't care about what order the elements are put into the map
- Use `LinkedHashMap` when you need to maintain insertion/access order
- Use `TreeMap` when you want the map sorted by key

- The `Map` interface does not provide an iterator
- To traverse a map you create an **entry set** of the mappings using the `entrySet()` method
- Each element in an entry set is a `String` that consists of the key and its value, separated by an equals sign

```
myMap = { "cs2"=45, "cs2s"=15, "cs4"=49 }  
Element 0: "cs2=45"  
Element 1: "cs2s=15"  
Element 2: "cs4=49"
```

MapDemo.java


```

public class MapDemo {
    public static void displayMapEntries(Map map) {
        Set entrySet = map.entrySet(); // Get an entry set for the map
        Iterator iter = entrySet.iterator(); // Get an iterator and loop over the map
        while (iter.hasNext()) {
            System.out.println("\t" + iter.next());
        }
    }

    public static void main(String args[]) {
        HashMap<String, Integer> hashMap = new HashMap<String, Integer>();
        System.out.println("Adding entries in this order:");
        System.out.println("\tcg1=35");           System.out.println("\tcs2s=15");
        System.out.println("\tcs2=45");           System.out.println("\tcs4=49");
        hashMap.put("cg1", 35);                   hashMap.put("cs2s", 15);
        hashMap.put("cs2", 45);                   hashMap.put("cs4", 49);
        System.out.println("Contents of hash map: ");    displayMapEntries(hashMap);
        LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<String, Integer>();
        linkedHashMap.put("cg1", 35);               linkedHashMap.put("cs2s", 15);
        linkedHashMap.put("cs2", 45);               linkedHashMap.put("cs4", 49);
        System.out.println("Contents of linked hash map: ");    displayMapEntries(linkedHashMap);
        TreeMap<String, Integer> treeMap = new TreeMap<String, Integer>();
        treeMap.put("cg1", 35);
        treeMap.put("cs2s", 15);
        treeMap.put("cs2", 45);
        treeMap.put("cs4", 49);
        System.out.println("Contents of tree map: ");    displayMapEntries(treeMap);
    } // main
} // MapDemo

```

1/17/2006
Java Collection Framework
49



MapDemo Output

OUTPUT:

```

-----
Adding entries in this order:
    cg1=35
    cs2s=15
    cs2=45
    cs4=49
Contents of hash map:
    cs4=49
    cs2=45
    cs2s=15
    cg1=35
Contents of linked hash map:
    cg1=35
    cs2s=15
    cs2=45
    cs4=49
Contents of tree map:
    cg1=35
    cs2=45
    cs2s=15
    cs4=49

```

1/17/2006
Java Collection Framework
50

- Write a program that counts the occurrences of words in a file and displays the words and their occurrences in **descending** order of word frequency
- One approach is to use a hash map of words (string key) and their frequency (integer value)
- To sort the values in descending order, we'll need to use another collection to store (key, value) pairs which implements a custom `Comparator`

```
import java.util.*; // HashMap, HashMap
import java.io.*; // BufferedReader, FileReader, IOException
public class WordCount {
    public static void main(String args[]) throws IOException {
        if (args.length != 1) {
            System.err.println("Usage: java WordCount text-file");
            System.exit(-1);
        }
        String filename = args[0];
        BufferedReader input = new BufferedReader(new FileReader(filename));
        String line;
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        // read in all the lines from the file
        while ((line = input.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(line, " ,.-");
            // process each word wrt the map
            while (st.hasMoreTokens()) {
                String word = st.nextToken();
                if (map.get(word) != null) {
                    // another occurrence of an existing word
                    int count = map.get(word);
                    count++;
                    map.put(word, count);
                } else {
                    // first occurrence of this word
                    map.put(word, 1);
                }
            }
        }
    }
}
```



WordCount.java

```

System.out.println("Unsorted map: " + map);
System.out.println();

// Now that all the words/counts are in the map, sort them
// based on frequency. To do this, use WordOccurrence.

// Create an array list to hold the WordOccurrence objects.
ArrayList<WordOccurrence> list = new ArrayList<WordOccurrence>(map.size());

// Get the set of words from the map
Set<String> words = map.keySet();

// Loop over the words in the map
Iterator<String> iter = words.iterator();
while (iter.hasNext()) {
    String word = iter.next();
    // Create a WordOccurrence object for each word/count pair
    WordOccurrence pair = new WordOccurrence(word, map.get(word));

    // Add it into the list
    list.add(pair);
}

// sort the list using the WordOccurrence (which implements Comparable)
Collections.sort(list);
// Print out the elements in the list that are now ordered by frequency
System.out.println("Words sorted by frequency: " + list);
} // main
} // WordCount
1/17/2006

```

Java Collection Framework

53



WordOccurrence.java

```

public class WordOccurrence implements Comparable<WordOccurrence> {
    private String word;           private int count;
    /** Construct the object
     * @param word the word
     * @param count the frequency count for the word
     */
    public WordOccurrence(String word, int count) {
        this.word = word;
        this.count = count;
    }
    /** Compare one object to another by descending frequency
     * @param other the other object
     */
    public int compareTo(WordOccurrence other) {
        if (other.count > count) {
            return 1;
        } else if (count == other.count) {
            return word.compareTo(other.word);
        } else {
            return -1;
        }
    } // compareTo
    /** Print a string representation of the object
     */
    public String toString() {
        return word + " = " + count;
    }
} // WordOccurrence

```

1/17/2006

Java Collection Framework

54

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way - in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

```
% java WordCount dickens.txt
```

Unsorted map: {only=1, like=1, some=1, superlative=1, noisiest=1, being=1, authorities=1, that=1, or=1, all=2, so=1, in=2, for=2, the=14, of=12, other=1, were=2, Light=1, times=2, hope=1, Heaven=1, season=2, incredulity=1, wisdom=1, it=9, present=1, short=1, good=1, winter=1, epoch=2, It=1, everything=1, direct=2, age=2, way=1, despair=1, before=2, comparison=1, degree=1, best=1, worst=1, received=1, period=2, evil=1, we=4, Darkness=1, foolishness=1, its=2, on=1, nothing=1, was=11, insisted=1, to=1, far=1, spring=1, us=2, had=2, belief=1, going=2}

Words sorted by frequency: [the = 14, of = 12, was = 11, it = 9, we = 4, age = 2, all = 2, before = 2, direct = 2, epoch = 2, for = 2, going = 2, had = 2, in = 2, its = 2, period = 2, season = 2, times = 2, us = 2, were = 2, Darkness = 1, Heaven = 1, It = 1, Light = 1, authorities = 1, being = 1, belief = 1, best = 1, comparison = 1, degree = 1, despair = 1, everything = 1, evil = 1, far = 1, foolishness = 1, good = 1, hope = 1, incredulity = 1, insisted = 1, like = 1, noisiest = 1, nothing = 1, on = 1, only = 1, or = 1, other = 1, present = 1, received = 1, short = 1, so = 1, some = 1, spring = 1, superlative = 1, that = 1, to = 1, way = 1, winter = 1, wisdom = 1, worst = 1]



- The previous example showed how to sort elements in a collection using the `Collections` class
- This class contains static methods for operating on collections and maps
 - Filling a list with an object
 - Sorting a list
 - Reversing a list



```
public class TestCollections {
    public static void main(String args[]) {
        List<String> list = Collections.nCopies(3, "red"); // Create a list of three strings
        // Create an array list of three elements
        ArrayList<String> arrayList = new ArrayList<String>(list);
        System.out.println("The initial list is: " + arrayList);
        Collections.fill(arrayList, "yellow");
        System.out.println("After filling with yellow: " + arrayList);
        // Add three new elements to the end of the list
        arrayList.add("white");
        arrayList.add("black");
        arrayList.add("orange");
        System.out.println("After adding new elements: " + arrayList);
        // Shuffle the list
        Collections.shuffle(arrayList);
        System.out.println("After shuffling: " + arrayList);
        // Reverse the list
        Collections.reverse(arrayList);
        System.out.println("After reversing: " + arrayList);
        // Find the minimum and maximum elements
        System.out.println("Minimum element: " + Collections.min(arrayList));
        System.out.println("Maximum element: " + Collections.max(arrayList));
        // Sort the list
        Collections.sort(arrayList);
        System.out.println("After sorting: " + arrayList);
        // Find an element in the list
        System.out.println("Binary search for white: " + Collections.binarySearch(arrayList, "white"));
    } // main
} // TestCollections
```



TestCollections Output

OUTPUT:

The initial list is: [red, red, red]

After filling with yellow: [yellow, yellow, yellow]

After adding new elements: [yellow, yellow, yellow, white, black, orange]

After shuffling: [white, yellow, yellow, black, orange, yellow]

After reversing: [yellow, orange, black, yellow, yellow, white]

Minimum element: black

Maximum element: yellow

After sorting: [black, orange, white, yellow, yellow, yellow]

Binary search for white: 2