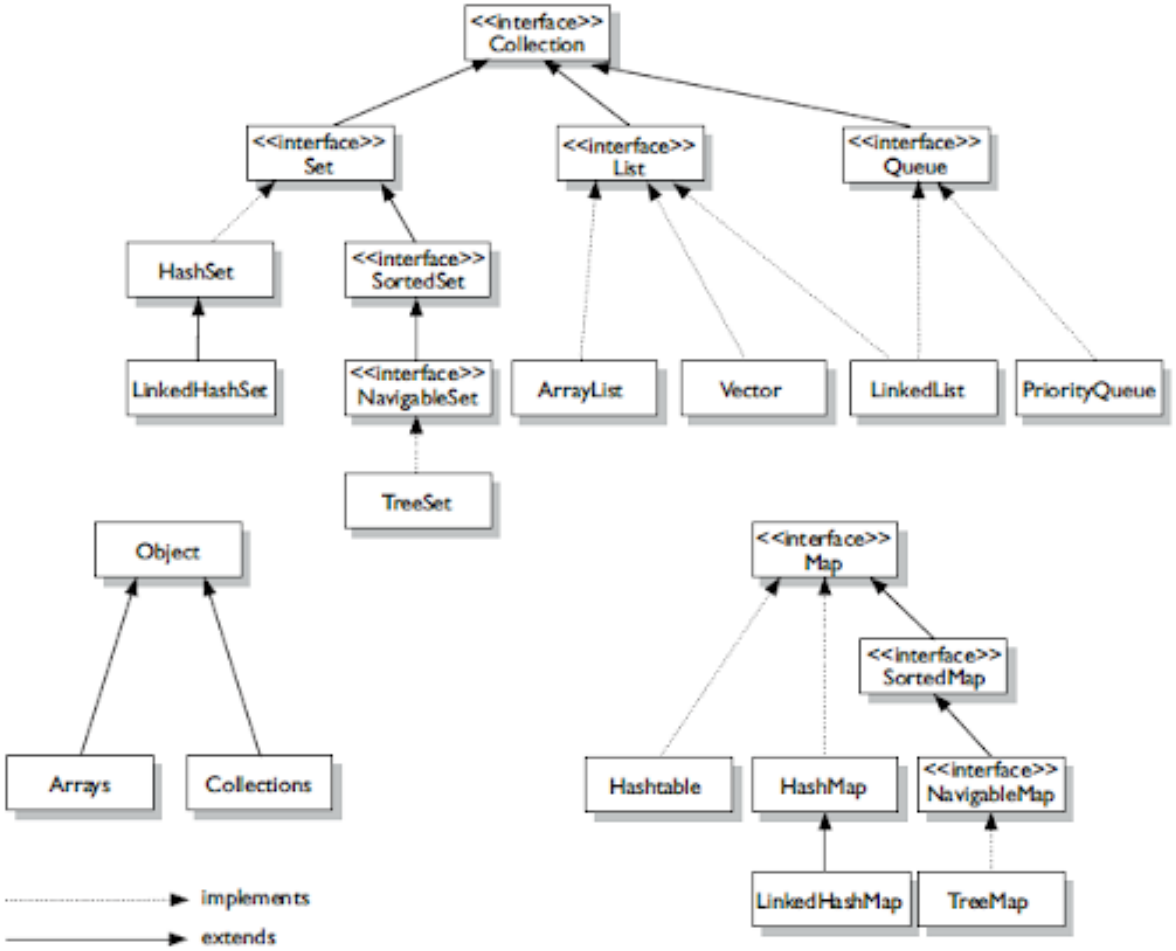
BBM 102 – Introduction to Programming II

Collections Framework

Today

- The **`java.util.Arrays`** class
- Java Collection Framework
  - java.util.Collection interface
  - java.util.List interface
    - java.util.ArrayList class
  - java.util.Set interface
    - java.util.HashSet class
  - java.util.Map interface
    - java.util.HashMap class
    - java.util.Hashtable class
    - java.util.Properties class



java.util.Arrays

- Provides high-level static methods for manipulating arrays, such as:
  - **sort** for sorting an array,
  - **binarySearch** for searching a sorted array,
  - **equals** for comparing arrays
  - **fill** for placing values into an array

Arrays – Example

```
 3  import java.util.Arrays;
 4
 5  public class UsingArrays
 6  {
 7     private int intArray[] = { 1, 2, 3, 4, 5, 6 };
 8     private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
 9     private int filledIntArray[], intArrayCopy[];
10
11     // constructor initializes arrays
12     public UsingArrays()
13     {
14        filledIntArray = new int [ 10 ]; // create int array with 10 elements
15        intArrayCopy = new int [ intArray.length ];
16
17        Arrays.fill( filledIntArray, 7 ); // fill with 7s
18        Arrays.sort( doubleArray ); // sort doubleArray ascending
19
20        // copy array intArray into array intArrayCopy
21        System.arraycopy( intArray, 0, intArrayCopy,
22           0, intArray.length );
23     } // end UsingArrays constructor
```

Arrays – Example (continued)

```
47      // find value in array intArray
48      public int searchForInt( int value )
49      {
50         return Arrays.binarySearch( intArray, value );
51      } // end method searchForInt
52
53      // compare array contents
54      public void printEquality()
55      {
56         boolean b = Arrays.equals( intArray, intArrayCopy );
57         System.out.printf( "intArray %s intArrayCopy\n",
58            ( b ? "==" : "!=" ) );
59
60         b = Arrays.equals( intArray, filledIntArray );
61         System.out.printf( "intArray %s filledIntArray\n",
62            ( b ? "==" : "!=" ) );
63      } // end method printEquality
```

Arrays – Example (continued)

```
26      public void printArrays()
27      {
28         System.out.print( "doubleArray: " );
29         for ( double doubleValue : doubleArray )
30            System.out.printf( "%.1f ", doubleValue );
31
32         System.out.print( "\nintArray: " );
33         for ( int intValue : intArray )
34            System.out.printf( "%d ", intValue );
35
36         System.out.print( "\nfilledIntArray: " );
37         for ( int intValue : filledIntArray )
38            System.out.printf( "%d ", intValue );
39
40         System.out.print( "\nintArrayCopy: " );
41         for ( int intValue : intArrayCopy )
42            System.out.printf( "%d ", intValue );
43
44         System.out.println( "\n" );
45      } // end method printArrays
```

Arrays – Example (continued)

```
65     public static void main( String args[] )
66     {
67        UsingArrays usingArrays = new UsingArrays();
68
69        usingArrays.printArrays();
70        usingArrays.printEquality();
71
72        int location = usingArrays.searchForInt( 5 );
73        if ( location >= 0 )
74           System.out.printf(
75              "Found 5 at element %d in intArray\n", location );
76        else
77           System.out.println( "5 not found in intArray" );
78
79        location = usingArrays.searchForInt( 8763 );
80        if ( location >= 0 )
81           System.out.printf(
82              "Found 8763 at element %d in intArray\n", location );
83        else
84           System.out.println( "8763 not found in intArray" );
85     } // end main
86  } // end class UsingArrays
```

Java Collections Framework

- Group of objects can be kept in an array, but arrays are not feasible when number of objects increase or decrease during the execution of the program
- The ***Java Collections Framework*** is a collection of interfaces and classes that may be used to manipulate groups of objects
- The classes implemented in the ***Java Collections Framework*** serve as reusable data structures and include algorithms for common tasks such as sorting or searching
- The framework uses parameterized classes so you can use them with the classes of your choice
- The framework is largely contained in the package **`java.util`**

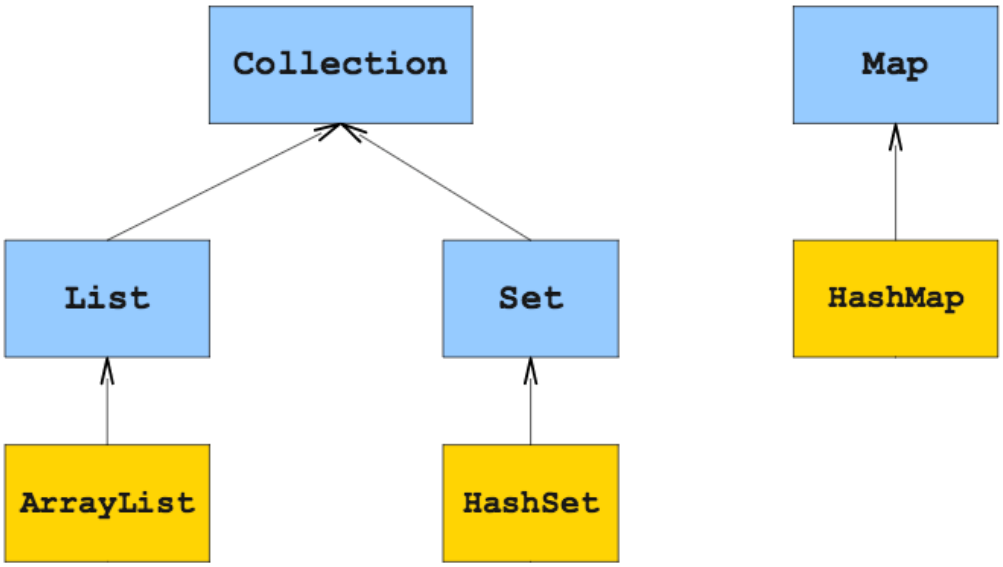
Basic collections

The three major collections in Java:

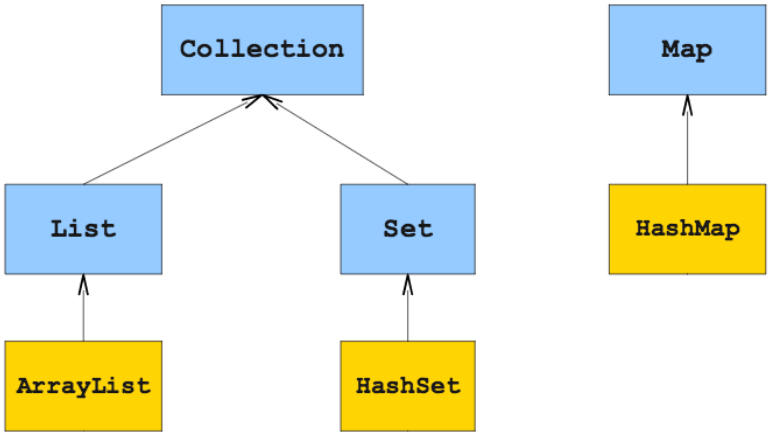
- ***Set***: A collection of elements that is guaranteed to contain no duplicates.
- ***List***: An ordered collection of elements, often accessed by integer indexes or by iteration.
- ***Map***: A collection of key/value pairs in which each key is associated with a corresponding value.

java.util.Collection interface

- It is the highest level of Java's framework for collection classes
- It describes the basic operations that all collection classes (except **`Map`**) should implement (listed in the next page)
- The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific sub-interfaces like Set and List as in the example diagram below:



What is a Set?



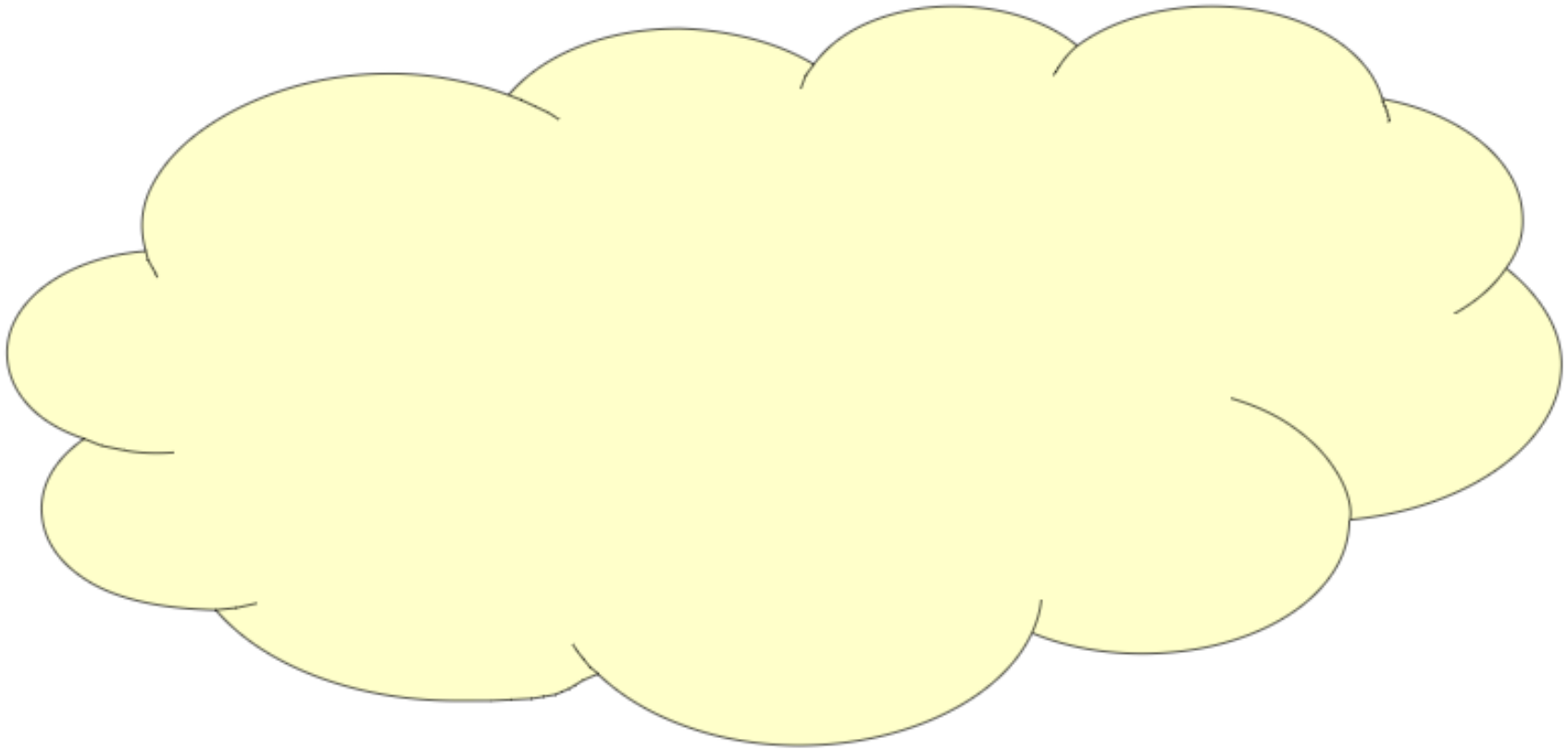
java.util.Set interface

- A **Set** is a **Collection** that contains unique elements (i.e., no duplicate elements).
- The collections framework contains several **Set** implementations, including **HashSet**
- Major operations are:
  - Adding an element
  - Removing an element
  - Checking whether an element exists

Basic Set Operations

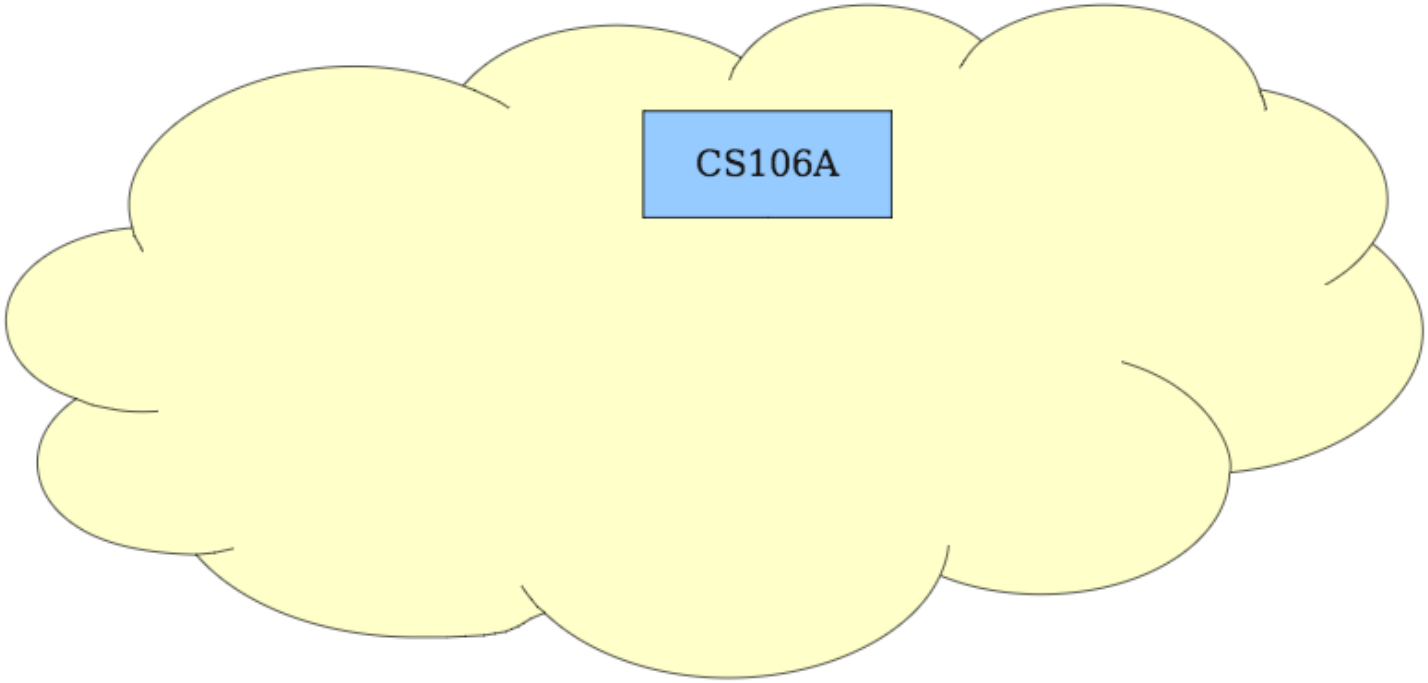
- To insert an item:
  - **set.add(value)**
- To check whether a value exists
  - **set.contains(value)**
- To remove an item
  - **set.remove(value)**

Defining a Set



```
HashSet<String> mySet = new HashSet<String>();
```

Adding Elements to Set

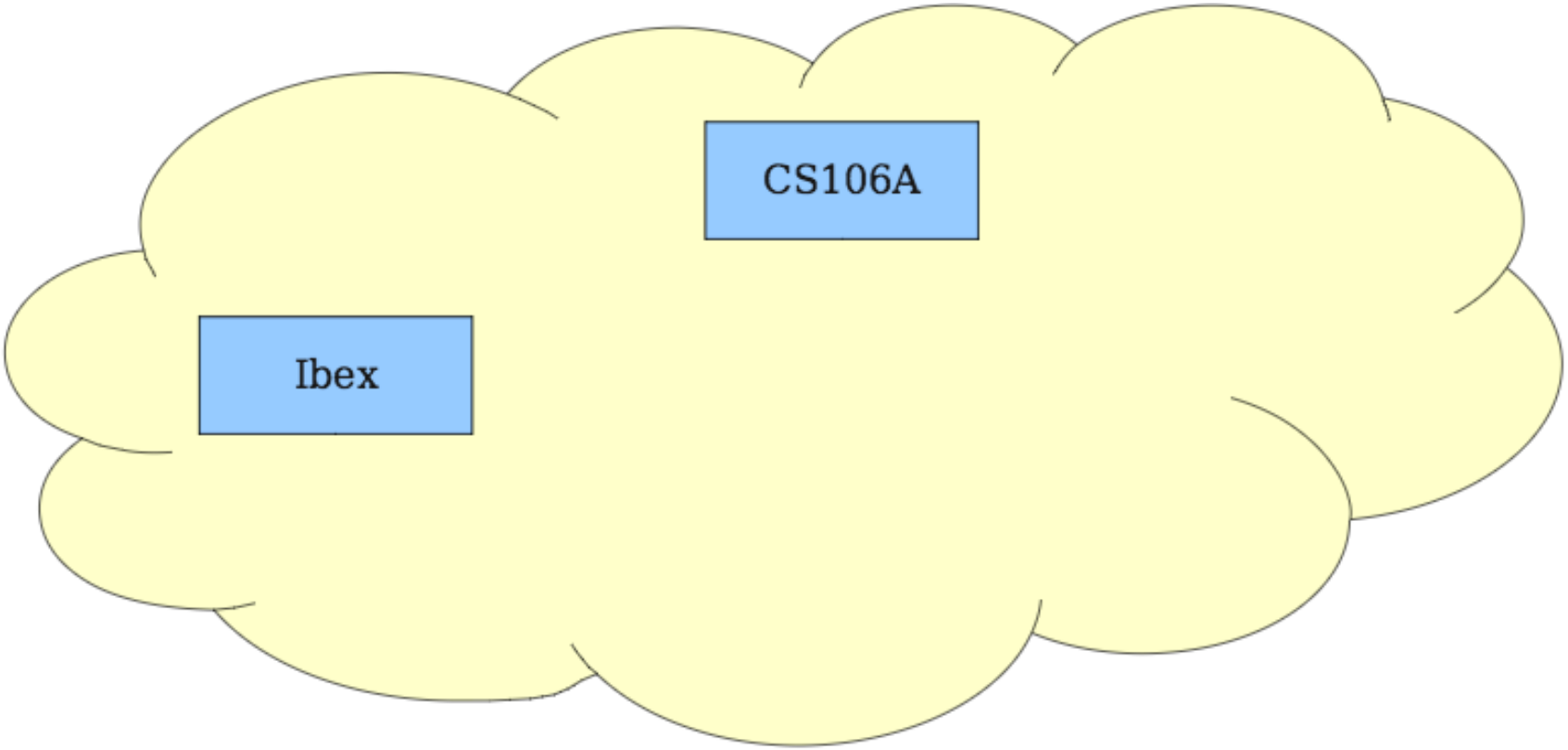



```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
```

To add a value to a **HashSet**, use the syntax

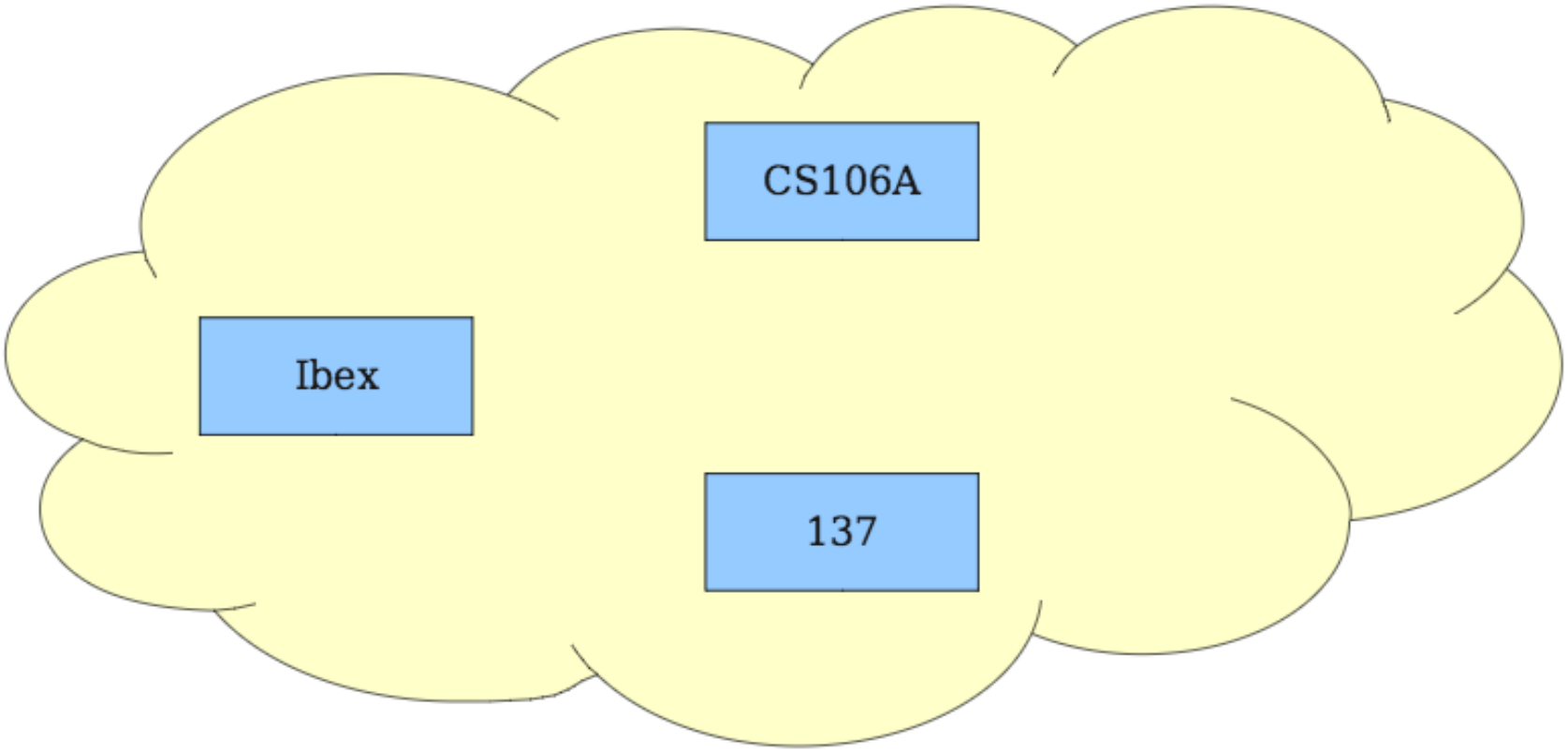
***set*.add(*value*)**

Adding Elements to Set (continued)



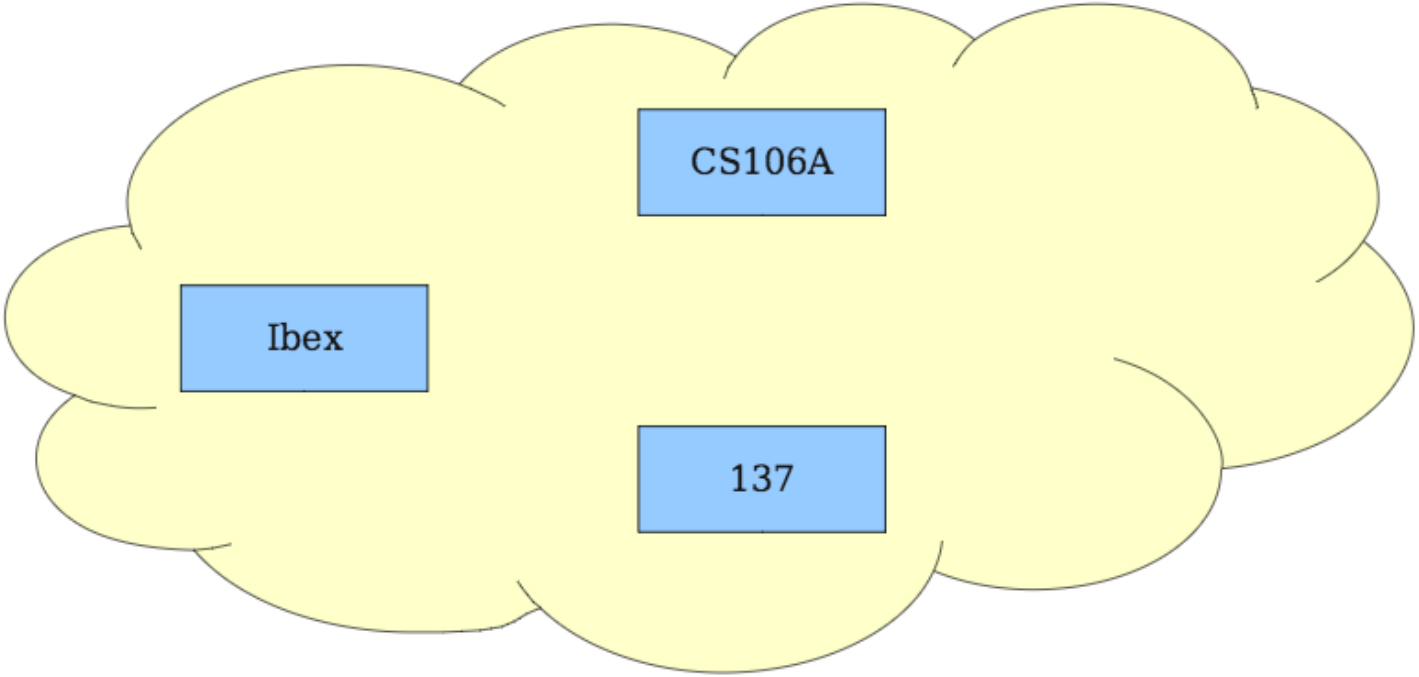

```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
```

Adding Elements to Set (continued)




```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
```

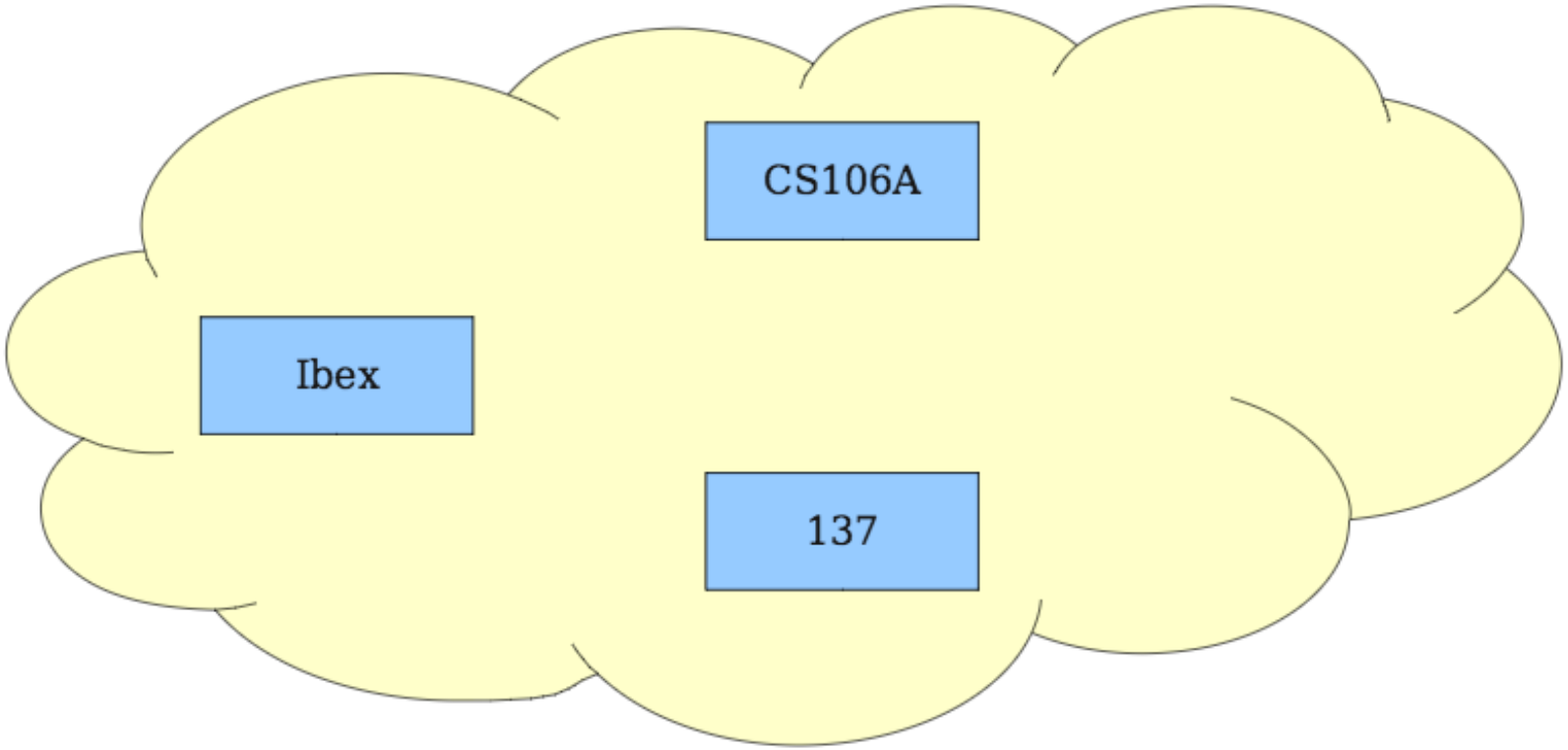
Adding Elements to Set (continued)




```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");
```

If you **add** a value pair where the value exists, nothing happens.

Checking Elements in Set (continued)




```
HashSet<String> mySet = new HashSet<String>();
mySet.add("CS106A");
mySet.add("Ibex");
mySet.add("137");
mySet.add("CS106A");

mySet.contains("Ibex");
```

HashSet - Example

```java
public class Test
{
    public static void main(String[]args)
    {
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet usind add()
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add(null);
        h.add("India");// adding duplicate elements

        // Displaying the HashSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing items from HashSet using remove()
        h.remove("Australia");
        System.out.println("List after removing Australia:"+h);

        // Iterating over hash set items
        System.out.println("Iterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

```
$java -Xmx128M -Xms16M Test

[null, South Africa, Australia, India]
List contains India or not:true
List after removing Australia:[null, South Africa, India]
Iterating over list:
null
South Africa
India
```

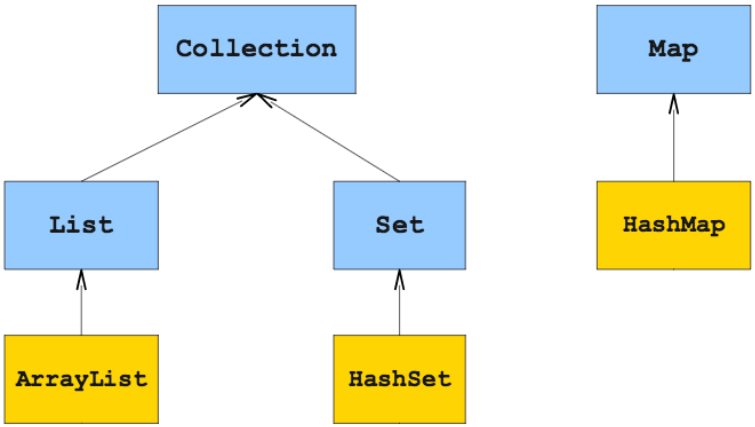
HashSet - Example

```
 3  import java.util.List;
 4  import java.util.Arrays;
 5  import java.util.HashSet;
 6  import java.util.Set;
 7  import java.util.Collection;
 8
 9  public class SetTest
10  {
11     private static final String colors[] = { "red", "white", "blue",
12        "green", "gray", "orange", "tan", "white", "cyan",
13        "peach", "gray", "orange" };
14
15     // create and output ArrayList
16     public SetTest()
17     {
18        List< String > list = Arrays.asList( colors );
19        System.out.printf( "ArrayList: %s\n", list );
20        printNonDuplicates( list );
21     } // end SetTest constructor
```

HashSet – Example (continued)

```
23      // create set from array to eliminate duplicates
24      private void printNonDuplicates( Collection< String > collection )
25      {
26         // create a HashSet
27         Set< String > set = new HashSet< String >( collection );
28
29         System.out.println( "\nNonduplicates are: " );
30
31         for ( String s : set )
32            System.out.printf( "%s ", s );
33
34         System.out.println();
35      } // end method printNonDuplicates
36
37      public static void main( String args[] )
38      {
39         new SetTest();
40      } // end main
41   } // end class SetTest
```

What is a List?



java.util.List interface

- A List (sometimes called a **sequence**) is an ordered Collection
- It can contain duplicate elements
- Like array indices, List indices are zero based (i.e., the first element's index is zero)
- In addition to the methods inherited from Collection, List provides methods for;
  - manipulating elements via their indices
  - manipulating a specified range of elements
  - searching for elements
  - getting a **ListIterator** to access the elements.

Iterators

- To visit every element of a collection, you can use the "for each" loop:

```
for(ElemType elem: collection){
    …
}
```

- Alternatively, you can use an **iterator**, an object whose job is to walk over the elements of a collection.
- The iterator has two commands:
  - **hasNext()** returns true if there are more items to visit.
  - **next()** returns the next item and moves the iterator to the next position.

ArrayList and Iterator - Example

```
 3  import java.util.List;
 4  import java.util.ArrayList;
 5  import java.util.Collection;
 6  import java.util.Iterator;
 7
 8  public class CollectionTest
 9  {
10     private static final String[] colors =
11        { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
12     private static final String[] removeColors =
13        { "RED", "WHITE", "BLUE" };
14
15     // create ArrayList, add Colors to it and manipulate it
16     public CollectionTest()
17     {
18        List< String > list = new ArrayList< String >();
19        List< String > removeList = new ArrayList< String >();
20
21        // add elements in colors array to list
22        for ( String color : colors )
23           list.add( color );
24
```

ArrayList and Iterator – Example (continued)

```
25         // add elements in removeColors to removeList
26         for ( String color : removeColors )
27            removeList.add( color );
28
29         System.out.println( "ArrayList: " );
30
31         // output list contents
32         for ( int count = 0; count < list.size(); count++ )
33            System.out.printf( "%s ", list.get( count ) );
34
35         // remove colors contained in removeList
36         removeColors( list, removeList );
37
38         System.out.println( "\n\nArrayList after calling removeColors: " );
39
40         // output list contents
41         for ( String color : list )
42            System.out.printf( "%s ", color );
43      } // end CollectionTest constructor
```

ArrayList and Iterator - Example (continued)

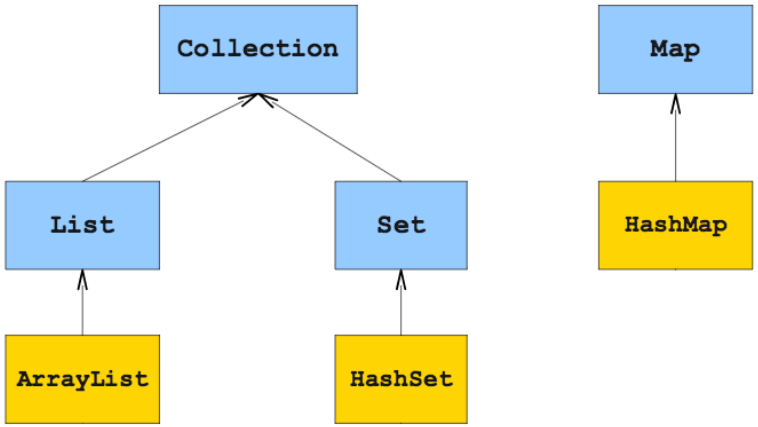
```
45     // remove colors specified in collection2 from collection1
46     private void removeColors(
47        Collection< String > collection1, Collection< String > collection2 )
48     {
49        // get iterator
50        Iterator< String > iterator = collection1.iterator();
51
52        // loop while collection has items
53        while ( iterator.hasNext() )
54
55           if ( collection2.contains( iterator.next() ) )
56              iterator.remove(); // remove current Color
57     } // end method removeColors
58
59     public static void main( String args[] )
60     {
61        new CollectionTest();
62     } // end main
63  } // end class CollectionTest
```

Is it possible to create an Array from ArrayList?

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ListToArray {

    public static void main(String[] args) {

        List<String> letters = new ArrayList<String>();

        // add example
        letters.add("A");
        letters.add("B");
        letters.add("C");

        //convert list to array
        String[] strArray = new String[letters.size()];
        strArray = letters.toArray(strArray);
        System.out.println(Arrays.toString(strArray)); //will print "[A, B, C]"
    }

}
```

```
$javac ListToArray.java

$java -Xmx128M -Xms16M ListToArray

[A, B, C]
```

What is a Map?

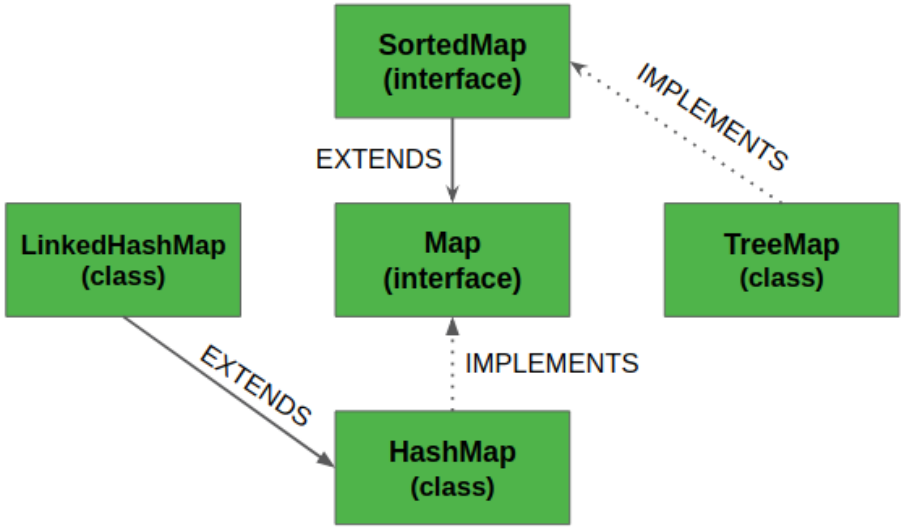


java.util.Map interface

- **Map**s associate keys to values and cannot contain duplicate keys (i.e., each key can map to only one value)
- **Map**s differ from **Set**s in that **Map**s contain keys and values, whereas **Set**s contain only values

Some notes about Maps

- A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the HashMap and LinkedHashMap, but some do not like the TreeMap.
- The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.



MAP Hierarchy in Java

HashMap - Example

```
 3  import java.util.StringTokenizer;
 4  import java.util.Map;
 5  import java.util.HashMap;
 6  import java.util.Set;
 7  import java.util.TreeSet;
 8  import java.util.Scanner;
 9
10  public class WordTypeCount
11  {
12     private Map< String, Integer > map;
13     private Scanner scanner;
14
15     public WordTypeCount()
16     {
17        map = new HashMap< String, Integer >(); // create HashMap
18        scanner = new Scanner( System.in ); // create scanner
19        createMap(); // create map based on user input
20        displayMap(); // display map content
21     } // end WordTypeCount constructor
```

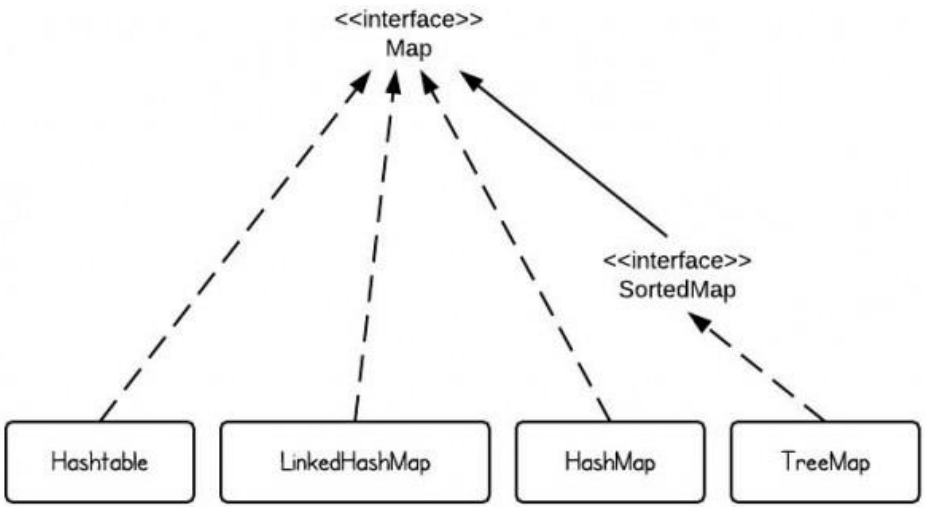
HashMap – Example (continued)

```
23      // create map from user input
24      private void createMap()
25      {
26         System.out.println( "Enter a string:" ); // prompt for user input
27         String input = scanner.nextLine();
28
29         // create StringTokenizer for input
30         StringTokenizer tokenizer = new StringTokenizer( input );
31
32         // processing input text
33         while ( tokenizer.hasMoreTokens() ) // while more input
34         {
35            String word = tokenizer.nextToken().toLowerCase(); // get word
36
37            // if the map contains the word
38            if ( map.containsKey( word ) ) // is word in map
39            {
40               int count = map.get( word ); // get current count
41               map.put( word, count + 1 ); // increment count
42            } // end if
43            else
44               map.put( word, 1 ); // add new word with a count of 1 to map
45          } // end while
46      } // end method createMap
```

HashMap – Example (continued)

```
48      // display map content
49      private void displayMap()
50      {
51         Set< String > keys = map.keySet(); // get keys
52
53         // sort keys
54         TreeSet< String > sortedKeys = new TreeSet< String >( keys );
55
56         System.out.println( "Map contains:\nKey\t\tValue" );
57
58         // generate output for each key in map
59         for ( String key : sortedKeys )
60            System.out.printf( "%-10s%10s\n", key, map.get( key ) );
61
62         System.out.printf(
63            "\nsize:%d\nisEmpty:%b\n", map.size(), map.isEmpty() );
64      } // end method displayMap
65
66      public static void main( String args[] )
67      {
68         new WordTypeCount();
69      } // end main
70   } // end class WordTypeCount
```

What is a Hashtable?



Hashtables

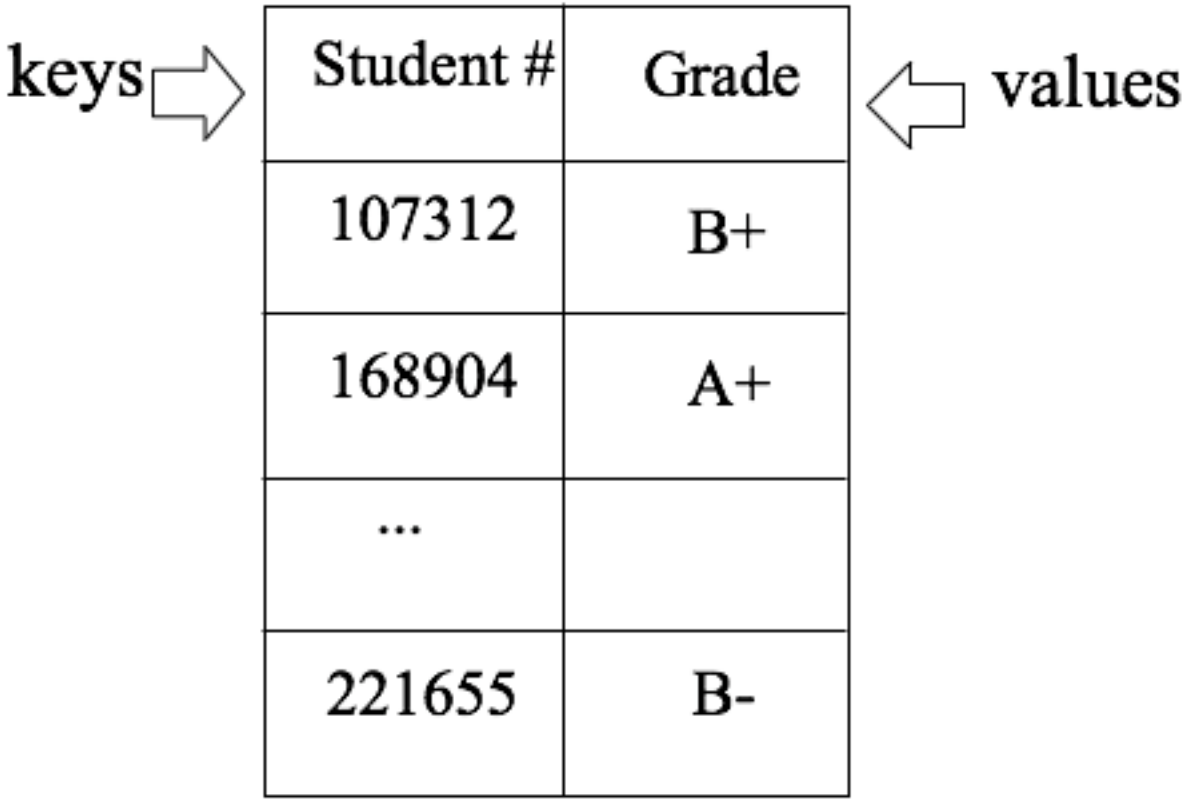
- This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. This means that, hashtable cannot allow null values or keys.
- To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

- It is similar to HashMap, but is synchronised.
- Hashtable stores key/value pair in hash table.

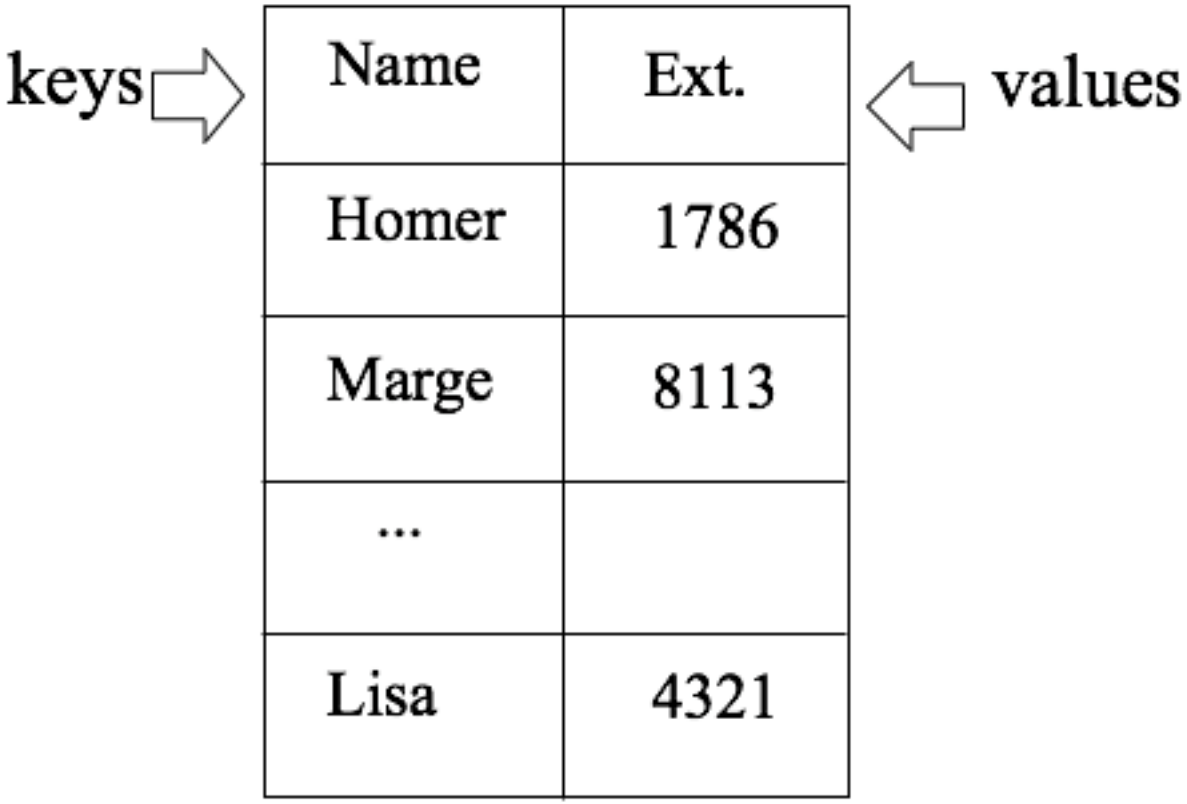
Hashtables

```
import java.util.*;
public class hashTabledemo {
    public static void main(String[] arg)
    {
        // creating a hash table
        Hashtable<Integer, String> h =  new Hashtable<Integer, String>();

        h.put(3, “Example");
        h.put(2, “of");
        h.put(1, “hashtable");

        System.out.println("value of key 2: "+ h.get(2));

        // clear hash table h
        h.clear();
        // checking hash table h
        System.out.println("after clearing: " + h);
    }
}
```

Hashtable Example

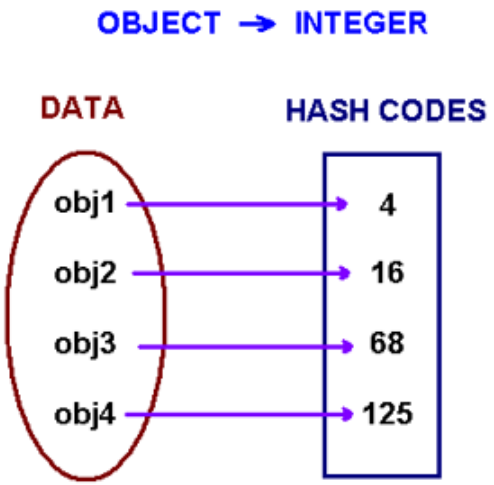


Hashtable Example (continued)



java.util.Properties class

- A **Properties** object is a *persistent* **Hashtable** that
  - stores key-value pairs of strings
  - assumes that you use methods **setProperty** and **getProperty** to manipulate the table rather than inherited **Hashtable** methods put and get.
  - A common use of Properties objects is to maintain application-configuration data or user preferences for applications



Properties - Example

```
 3   import java.io.FileOutputStream;
 4   import java.io.FileInputStream;
 5   import java.io.IOException;
 6   import java.util.Properties;
 7   import java.util.Set;
 8
 9   public class PropertiesTest
10   {
11      private Properties table;
12
13      // set up GUI to test Properties table
14      public PropertiesTest()
15      {
16         table = new Properties(); // create Properties table
17
18         // set properties
19         table.setProperty( "color", "blue" );
20         table.setProperty( "width", "200" );
```

Properties – Example (continued)

```
22          System.out.println( "After setting properties" );
23          listProperties(); // display property values
24
25          // replace property value
26          table.setProperty( "color", "red" );
27
28          System.out.println( "After replacing properties" );
29          listProperties(); // display property values
30
31          saveProperties(); // save properties
32
33          table.clear(); // empty table
34
35          System.out.println( "After clearing properties" );
36          listProperties(); // display property values
37
38          loadProperties(); // load properties
```

Properties – Example (continued)

```
40          // get value of property color
41          Object value = table.getProperty( "color" );
42
43          // check if value is in table
44          if ( value != null )
45             System.out.printf( "Property color's value is %s\n", value );
46          else
47             System.out.println( "Property color is not in table" );
48       } // end PropertiesTest constructor
```

Properties – Example (continued)

```
50      // save properties to a file
51      public void saveProperties()
52      {
53         // save contents of table
54         try
55         {
56            FileOutputStream output = new FileOutputStream( "props.dat" );
57            table.store( output, "Sample Properties" ); // save properties
58            output.close();
59            System.out.println( "After saving properties" );
60            listProperties();
61         } // end try
62         catch ( IOException ioException )
63         {
64            ioException.printStackTrace();
65         } // end catch
66      } // end method saveProperties
```

Properties – Example (continued)

```
69      public void loadProperties()
70      {
71         // load contents of table
72         try
73         {
74            FileInputStream input = new FileInputStream( "props.dat" );
75            table.load( input ); // load properties
76            input.close();
77            System.out.println( "After loading properties" );
78            listProperties(); // display property values
79         } // end try
80         catch ( IOException ioException )
81         {
82            ioException.printStackTrace();
83         } // end catch
84      } // end method loadProperties
```

Properties – Example (continued)

```
86       // output property values
87       public void listProperties()
88       {
89          Set< Object > keys = table.keySet(); // get property names
90
91          // output name/value pairs
92          for ( Object key : keys )
93          {
94             System.out.printf(
95                "%s\t%s\n", key, table.getProperty( ( String ) key ) );
96          } // end for
97
98          System.out.println();
99       } // end method listProperties
100
101      public static void main( String args[] )
102      {
103         new PropertiesTest();
104      } // end main
105   } // end class PropertiesTest
```

Summary

- Arrays are used for a group of objects, but maintanence is difficult when the size of the group changes during the execution of the program
- Java Collection Framework includes many interfaces and classes to easily manage groups of objects
- The interfaces and classes of the framework are mostly in **java.util** package
- Various interfaces include methods for special algorithms of various data structures
- Framework includes various implementations of these interfaces
- It is possible to implement or extend the interfaces/classes of the framework for new/different implementations of new/different data structures

Acknowledgements

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Building Java Programs – A Back to Basics Approach, Stuart Reges and Marty Stepp, Addison Wesley, 2011
  - Stanford, Collections – lecture notes