

Collection Framework

Objective

- Collection Framework

- Core Interfaces

Java Collections Framework

- The **Java Collections Framework** is a collection of
  - interfaces and classes 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.

Collections Framework

Collections Framework is a unified architecture for managing collections

This framework is provided in the java.util package

Main Parts of Collections Framework

-Interfaces

- Core interfaces defining common functionality exhibited by collections

-Implementations

- Concrete classes of the core interfaces providing data structures

Collections Framework

Interfaces

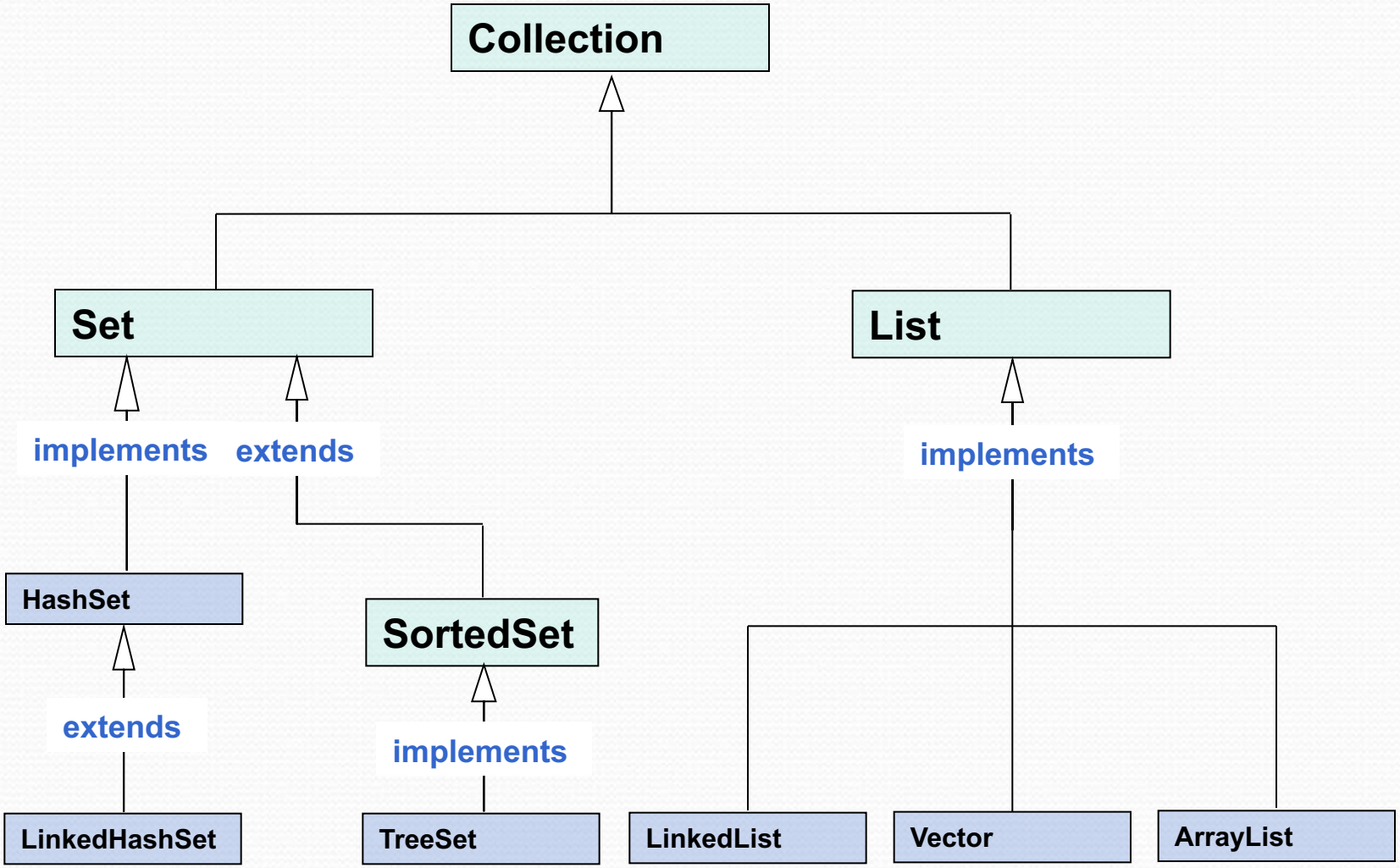
Core Interface	*Description*
`Collection`	specifies contract that all collections should implement
`Set`	defines functionality for a *set* of unique elements
`SortedSet`	defines functionality for a *set* where elements are sorted
`List`	defines functionality for an ordered *list* of non- unique elements
`Map`	defines functionality for mapping of unique keys to values
`SortedMap`	defines functionality for a *map* where its keys are sorted

Collections Framework Implementations

Set	List	Map
`HashSet`	**`ArrayList`**	**`HashMap`**
`LinkedHashSet`	**`LinkedList`**	**`LinkedHashMap`**
`TreeSet`	**`Vector`**	**`Hashtable`**
		`Tree Map`

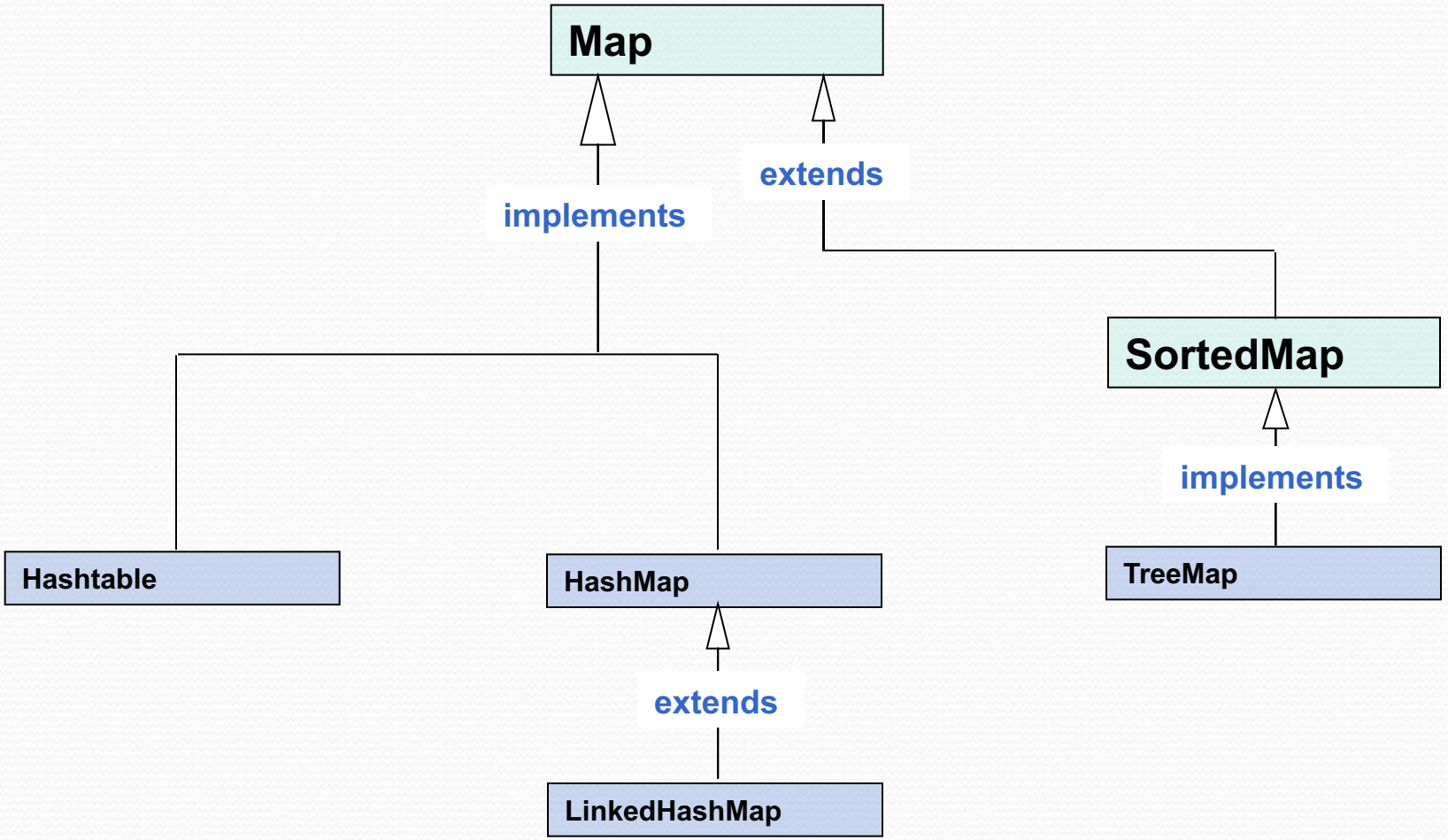
Note: Hashtable uses a lower-case “t”

Collections Hierarchy
Set and List



Collections Hierarchy

Map



Operations

Basic collection operations:

- Check if collection is empty
- Check if an object exists in collection.
- Retrieve an object from collection
- Add object to collection
- Remove object from collection
- Iterate collection and inspect each object

- Each operation has a corresponding method implementation for each collection type

Collections Methods

- Class `Collections` provides several high-performance algorithms for manipulating collection elements.
- The algorithms (Fig. 20.5) are implemented as `static` methods.
- Note that Collection is a base interface for most collection classes, whereas Collections is a utility class.

Collection : The root interface of Java Collections Framework.

Collections : A utility class that is a member of the Java Collections Framework.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`.
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a `Collection`.
`frequency`	Calculates how many collection elements are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Fig. 20.5 | `Collections` methods.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.



Software Engineering Observation 20.4

The collections framework methods are polymorphic. That is, each can operate on objects that implement specific interfaces, regardless of the underlying implementations.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Lists

- A List (sometimes called a **sequence**) is an ordered Collection that can contain duplicate elements.
- Like array indices, List indices are zero based (i.e., the first element's index is zero).
- Interface List is implemented by several classes, including **ArrayList**, **LinkedList** and **Vector**

Array List

- An instance of ArrayList is created and named in the same way
- As objects of any class are created and named, except that you specify the base type using a different notation< >.
- Example

```
ArrayList<String>list = new ArrayList<String>(20);
```

Array List

- `ArrayList` collection
  - Similar to arrays
  - Dynamic resizing
    - They automatically increase their size at execution time to accommodate additional elements

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Introduction to Collections and Class `ArrayList`

- Java API provides several predefined data structures, called collections, used to store groups of related objects.
  - Each provides efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
  - Reduce application-development time.
- Arrays do not automatically change their size at execution time to accommodate additional elements.
- ArrayList<T> (package `java.util`) can dynamically change its size to accommodate more elements.
  - `T` is a placeholder for the type of element stored in the collection.
  - This is similar to specifying the type when declaring an array, except that only nonprimitive types can be used with these collection classes.
- Classes with this kind of placeholder that can be used with any type are called generic classes.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Introduction to Collections and Class `ArrayList` (Cont.)

- Figure 7.24 demonstrates some common `ArrayList` capabilities.
- An `ArrayList`'s capacity indicates how many items it can hold without growing.
- When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array.
  - This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added.
  - An `ArrayList` grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Method	Description
`add`	Adds an element to the end of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

Introduction to Collections and Class `ArrayList` (Cont.)

- Method add adds elements to the `ArrayList`.
  - One-argument version appends its argument to the end of the `ArrayList`.
  - Two-argument version inserts a new element at the specified position.
  - Collection indices start at zero.
- Method size returns the number of elements in the `ArrayList`.
- Method get obtains the element at a specified index.
- Method remove deletes an element with a specific value.
  - An overloaded version of the method removes the element at the specified index.
- Method contains determines if an item is in the `ArrayList`.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

```
1  // Fig. 7.24: ArrayListCollection.java
2  // Generic ArrayList<T> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
7     public static void main( String[] args )
8     {
9        // create a new ArrayList of Strings with an initial capacity of 10
10       ArrayList< String > items = new ArrayList< String >();
11
12       items.add( "red" ); // append an item to the list
13       items.add( 0, "yellow" ); // insert the value at index 0
14
15       // header
16       System.out.print(
17          "Display list contents with counter-controlled loop:" );
18
19       // display the colors in the list
20       for ( int i = 0; i < items.size(); i++ )
21          System.out.printf( " %s", items.get( i ) );
22
```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 1 of 3.)

```
23         // display colors using foreach in the display method
24         display( items,
25            "\nDisplay list contents with enhanced for statement:" );
26
27         items.add( "green" ); // add "green" to the end of the list
28         items.add( "yellow" ); // add "yellow" to the end of the list
29         display( items, "List with two new elements:" );
30
31         items.remove( "yellow" ); // remove the first "yellow"
32         display( items, "Remove first instance of yellow:" );
33
34         items.remove( 1 ); // remove item at index 1
35         display( items, "Remove second list element (green):" );
36
37         // check if a value is in the List
38         System.out.printf( "\"red\" is %sin the list\n",
39            items.contains( "red" ) ? "": "not " );
40
41         // display number of elements in the List
42         System.out.printf( "Size: %s\n", items.size() );
43      } // end main
44
```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 2 of 3.)

```
45      // display the ArrayList's elements on the console
46      public static void display( ArrayList< String > items, String header )
47      {
48         System.out.print( header ); // display header
49
50         // display each element in items
51         for ( String item : items )
52            System.out.printf( " %s", item );
53
54         System.out.println(); // display end of line
55      } // end method display
56   } // end class ArrayListCollection
```

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Fig. 7.24 | Generic `ArrayList<T>` collection demonstration. (Part 3 of 3.)

Code Example: ArrayListCollection

- See code

`ArrayList` and `Iterator`

- List method `add` adds an item to the end of a list.
- List method `size` returns the number of elements.
- List method `get` retrieves an individual element's value from the specified index.
- In Java, an iterator is an interface and is implemented by all collection classes. The Java collections framework is a group of classes and interfaces that implement reusable collection of data structures. The iterator method returns an object that implements the Iterator interface.
- Collection method iterator gets an `Iterator` for a `Collection`.
- Iterator- method hasNext determines whether a `Collection` contains more elements.
  - Returns `true` if another element exists and `false` otherwise.
- Iterator method next obtains a reference to the next element.
- `Collection` method contains determine whether a `Collection` contains a specified element.
- Iterator method remove removes the current element from a `Collection`.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Collection Interface

- See code fig 20.2

```
42      // remove colors specified in collection2 from collection1
43      private static void removeColors( Collection< String > collection1,
44         Collection< String > collection2 )
45      {
46         // get iterator
47         Iterator< String > iterator = collection1.iterator();
48
49         // loop while collection has items
50         while ( iterator.hasNext() )
51         {
52            if ( collection2.contains( iterator.next() ) )
53               iterator.remove(); // remove current Color
54         } // end while
55      } // end method removeColors
56   } // end class CollectionTest
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Fig. 20.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 3 of 3.)

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

ArrayList and Iterator

- New in Java SE 7: Type Inference with the <> Notation
  - Lines 14 and 21 specify the type stored in the `ArrayList` (that is, `String`) on the left and right sides of the initialization statements.
  - Java SE 7 supports type inferencing with the <> notation in statements that declare and create generic type variables and objects. For example, line 14 can be written as:

```
List< String > list = new ArrayList <>();
```

  - Java uses the type in angle brackets on the left of the declaration (that is, `String`) as the type stored in the `ArrayList` created on the right side of the declaration.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Method `sort`

- Method sort sorts the elements of a `List`
  - The elements must implement the Comparable interface.
  - The order is determined by the natural order of the elements' type as implemented by a `compareTo` method.
  - Method `compareTo` is declared in interface `Comparable` and is sometimes called the natural comparison method.
  - The `sort` call may specify as a second argument a Comparator object that determines an alternative ordering of the elements.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

- Class `Arrays` provides `static` method asList to view an array as a List collection.
  - A `List` view allows you to manipulate the array as if it were a list.
  - This is useful for adding the elements in an array to a collection and for sorting array elements.
- Any modifications made through the `List` view change the array, and any modifications made to the array change the `List` view.
- List method toArray gets an array from a `List` collection.

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Collection Method Sort

- See Fig 20.6 Code

```
Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]
Sorted array elements: [Clubs, Diamonds, Hearts, Spades]
```

Fig. 20.6 | `Collections` method `sort`. (Part 2 of 2.)

© Copyright 1992-2012 by Pearson Education, Inc.
All Rights Reserved.

Binary Search

- java.util.Collections.binarySearch() method is a java.util.Collections class method that returns position of an object in a sorted list.

```
// Returns index of key in sorted list sorted in // ascending order

 public static int binarySearch(List slist, T key)

 // Returns index of key in sorted list sorted in
// order defined by Comparator c.

 public static int binarySearch(List slist, T key, Comparator c)
```

If key is not present, the it returns "(-(insertion point) - 1)".
The insertion point is defined as the point at which the key would be inserted into the list

```java
import java.util.*;
public class GFG {
    public static void main(String[] args)
    {
        List<Integer> al = new ArrayList<Integer>();
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(10);
        al.add(20);

        // 10 is present at index 3.
        int index = Collections.binarySearch(al, 10);
        System.out.println(index);

        // 13 is not present. 13 would have been inserted
        // at position 4. So the function returns (-4-1)
        // which is -5.
        index = Collections.binarySearch(al, 13);
        System.out.println(index);
    }
}
```