The Java Collections Framework

Definition

Set of interfaces, abstract and concrete classes that define common abstract data types in Java

- e.g. list, stack, queue, set, map

Part of the java.util package

Implementation

Extensive use of generic types, hash codes (Object.hashCode()) , and Comparable interface (compareTo(), e.g. for sorting)

Collection Interface

Defines common operations for sets and lists (‘unordered’ ops.)

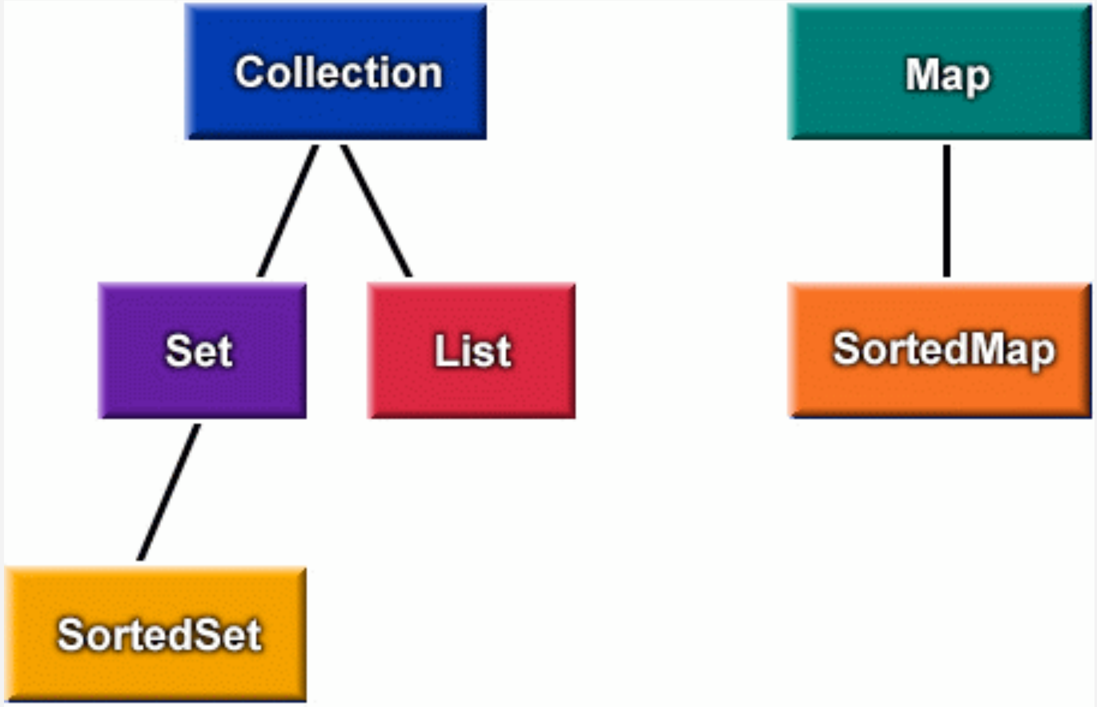
Maps

Represented by separate interfaces from list/set

(due to key/value relationship vs. a group of elements)

Java Collections Interfaces

(slide: Carl Reynolds)



Note: Some of the material on these slides was taken from the Java Tutorial at http://www.java.sun.com/docs/books/tutorial

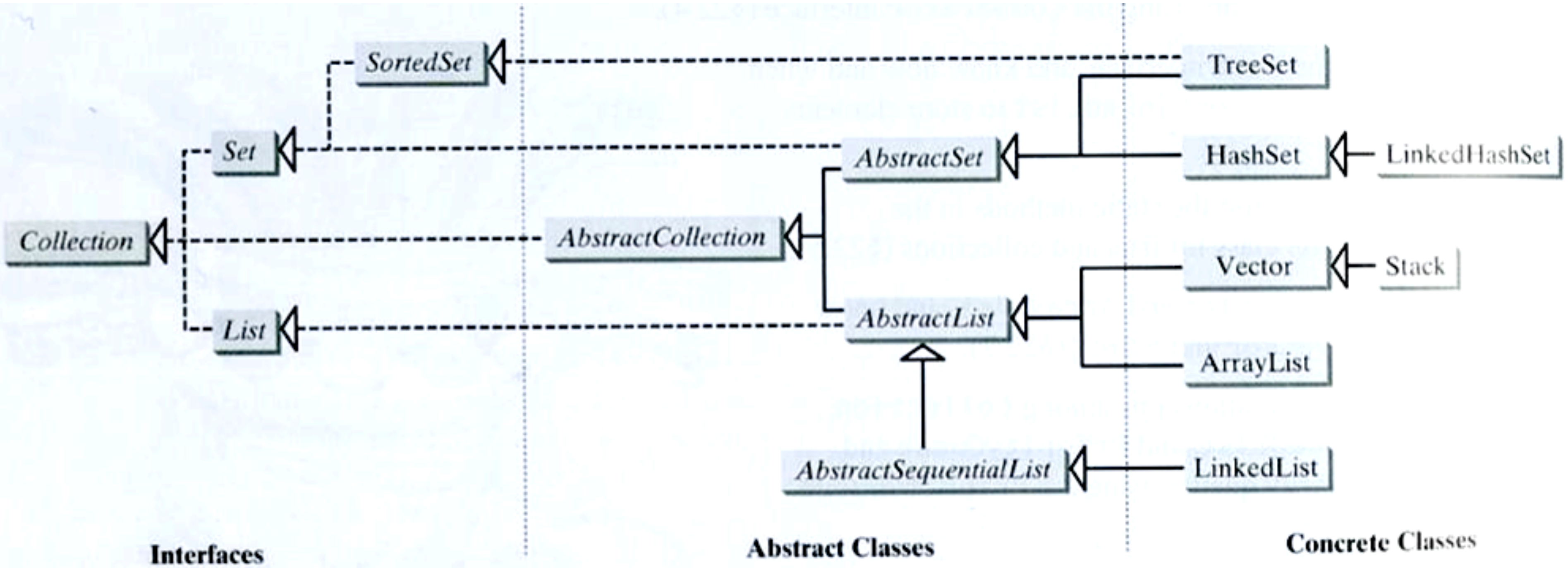


FIGURE 22.1 Set and List are subinterfaces of Collection.

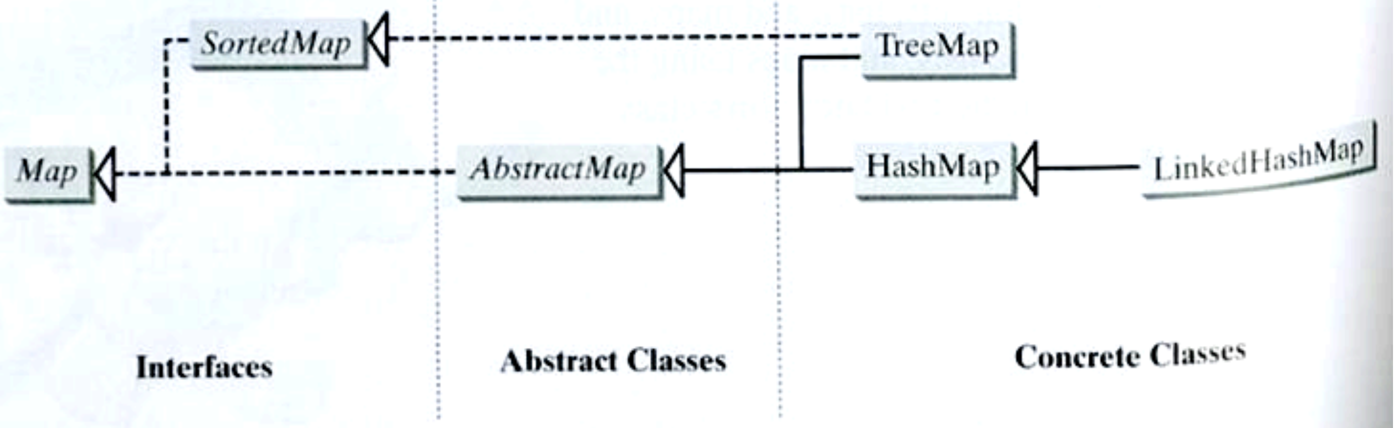


FIGURE 22.2 An instance of Map stores a group of objects and their associated keys.

Implementation Classes

(slide derived from: Carl Reynolds)

Interface	Implementation			
Set	HashSet		TreeSet	LinkedHashSet
List		ArrayList		LinkedList
Map	HashMap		TreeMap	LinkedHashMap

Note: When writing programs use the interfaces rather than the implementation classes where you can: this makes it easier to change implementations of an ADT.

Notes on ‘Unordered’ Collections (Set, Map Implementations)

HashMap, HashSet

Hash table implementation of set/map

Use hash codes (integer values) to determine where set elements or (key,value) pairs are stored in the *hash table* (array)

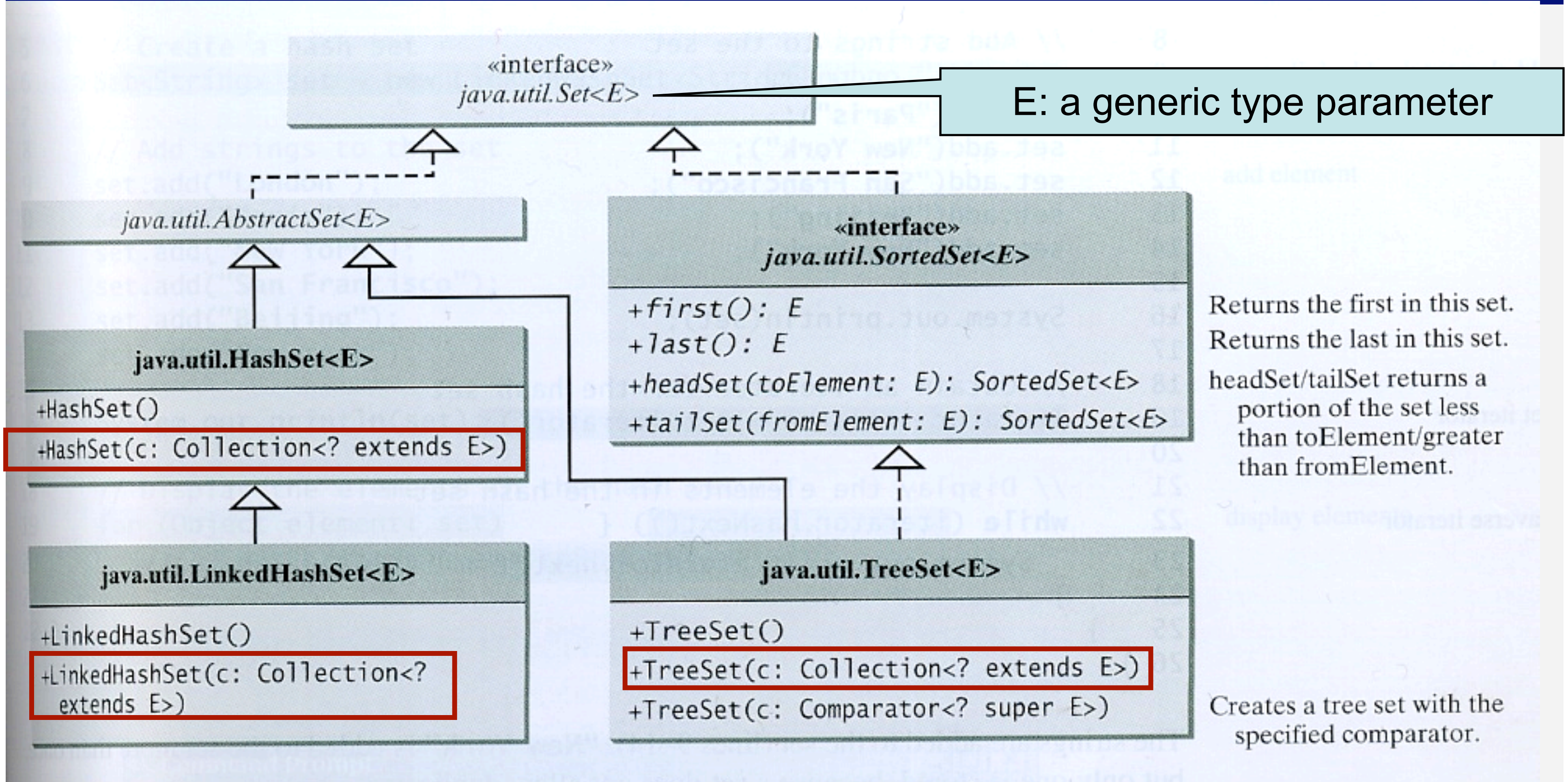
LinkedHashMap, LinkedHashSet

Provide support for arranging set elements or (key,value) pairs by order of insertion by adding a *linked list within the hash table elements*

TreeMap,TreeSet

Use binary search tree implementations to order set elements by value, or (key,value) pairs by key value

Sets in the Collections Framework



E: a generic type parameter

«interface»
java.util.Set<E>

java.util.AbstractSet<E>

«interface»
java.util.SortedSet<E>

```
+first(): E
+last(): E
+headSet(toElement: E): SortedSet<E>
+tailSet(fromElement: E): SortedSet<E>
```

Returns the first in this set.

Returns the last in this set.

headSet/tailSet returns a portion of the set less than toElement/greater than fromElement.

java.util.HashSet<E>

```
+HashSet()
+HashSet(c: Collection<? extends E>)
```

java.util.LinkedHashSet<E>

```
+LinkedHashSet()
+LinkedHashSet(c: Collection<?
 extends E>)
```

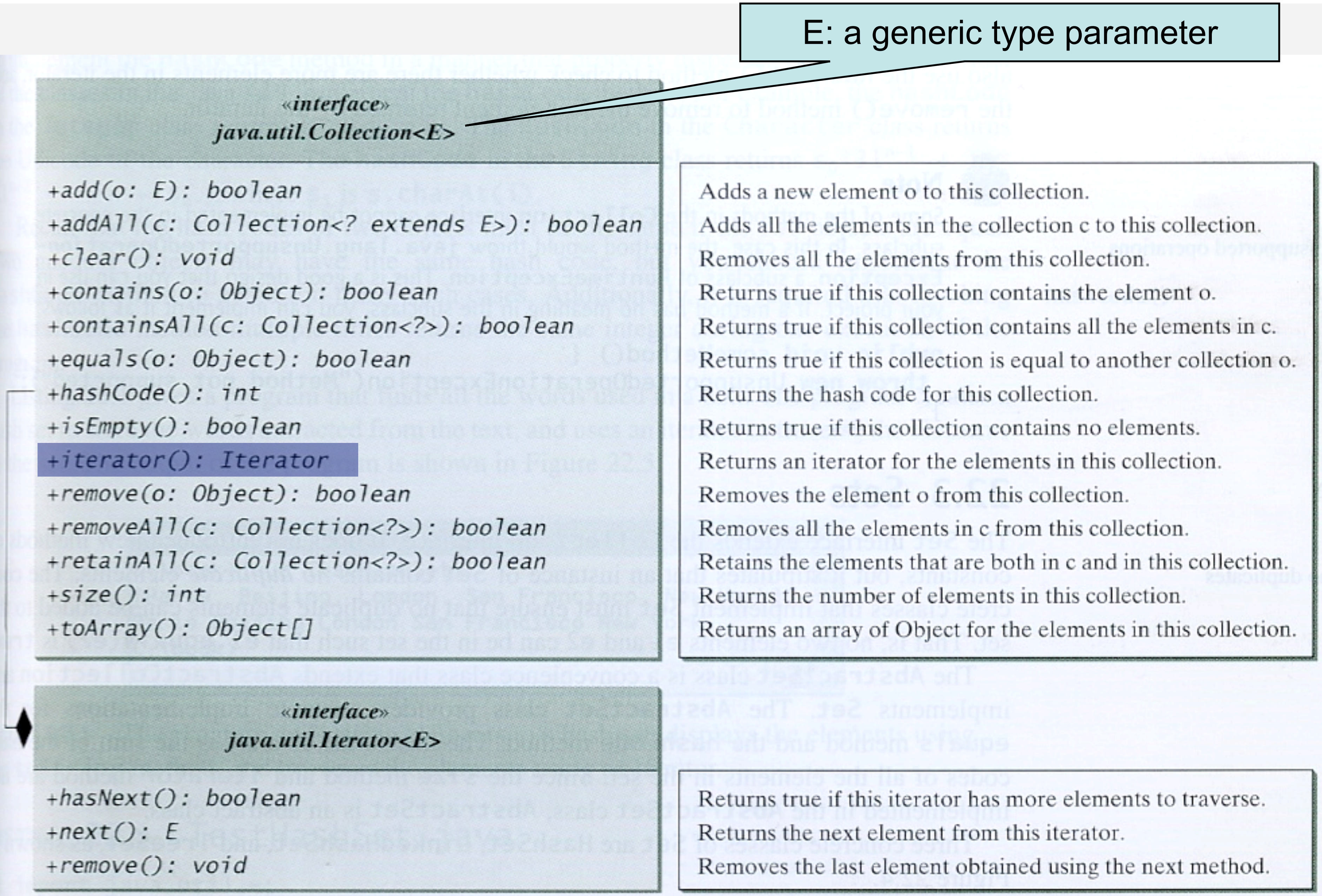
java.util.TreeSet<E>

```
+TreeSet()
+TreeSet(c: Collection<? extends E>)
+TreeSet(c: Comparator<? super E>)
```

Creates a tree set with the specified comparator.

FIGURE 22.4 The Java Collections Framework provides three concrete set classes.

E: a generic type parameter



«interface»
java.util.Collection<E>

+add(o: E): boolean	Adds a new element o to this collection.
+addAll(c: Collection<? extends E>): boolean	Adds all the elements in the collection c to this collection.
+clear(): void	Removes all the elements from this collection.
+contains(o: Object): boolean	Returns true if this collection contains the element o.
+containsAll(c: Collection<?>): boolean	Returns true if this collection contains all the elements in c.
+equals(o: Object): boolean	Returns true if this collection is equal to another collection o.
+hashCode(): int	Returns the hash code for this collection.
+isEmpty(): boolean	Returns true if this collection contains no elements.
+iterator(): Iterator	Returns an iterator for the elements in this collection.
+remove(o: Object): boolean	Removes the element o from this collection.
+removeAll(c: Collection<?>): boolean	Removes all the elements in c from this collection.
+retainAll(c: Collection<?>): boolean	Retains the elements that are both in c and in this collection.
+size(): int	Returns the number of elements in this collection.
+toArray(): Object[]	Returns an array of Object for the elements in this collection.

«interface»
java.util.Iterator<E>

+hasNext(): boolean	Returns true if this iterator has more elements to traverse.
+next(): E	Returns the next element from this iterator.
+remove(): void	Removes the last element obtained using the next method.

FIGURE 22.3 The `Collection` interface contains the methods for manipulating the elements in a collection, and each collection object contains an iterator for traversing elements in the collection.

HashSet
(Example: TestHashSet.java, Liang)

Methods:

Except for constructors, defined methods identical to Collection

Element Storage:

'Unordered,' but stored in a hash table according to their hash codes

**All elements are unique

Do not expect to see elements in the order you add them when you output them using toString().

Hash Codes

- Most classes in Java API override the hashCode() method in the Object class
- Need to be defined to properly disperse set elements in storage (i.e. throughout locations of the hash table)
- For two equivalent objects, hash codes must be the same

LinkedHashSet (example: TestLinkedHashSet.java)

Methods

Again, same as Collection Interface except for constructors

Addition to HashSet

– Elements in *hash table* contain an extra field defining order in which elements are added (as a linked list)

– List maintained by the class

Hash Codes

Notes from previous slide still apply (e.g. equivalent objects, equivalent hash codes)

Ordered Sets: TreeSet (example: TestTreeSet.java)

Methods

Add methods from *SortedSet* interface:

first(), last(), headSet(toElement: E), tailSet(fromElement: E)

Implementation

A binary search tree, such that either:

1. Objects (elements) implement the *Comparable* interface (compareTo()) ("natural order" of objects in a class), *or*
2. TreeSet is constructed using an object implementing the *Comparator* interface (compare()) to determine the ordering (permits comparing objects of the same or different classes, create different orderings)

One of these will determine the ordering of elements.

Notes

– It is faster to use a hash set to retrieve elements, as TreeSet keeps elements in a sorted order (making search necessary)

– Can construct a tree set using an existing collection (e.g. a hash set)

Iterator Interface

Purpose

Provides uniform way to traverse sets and lists

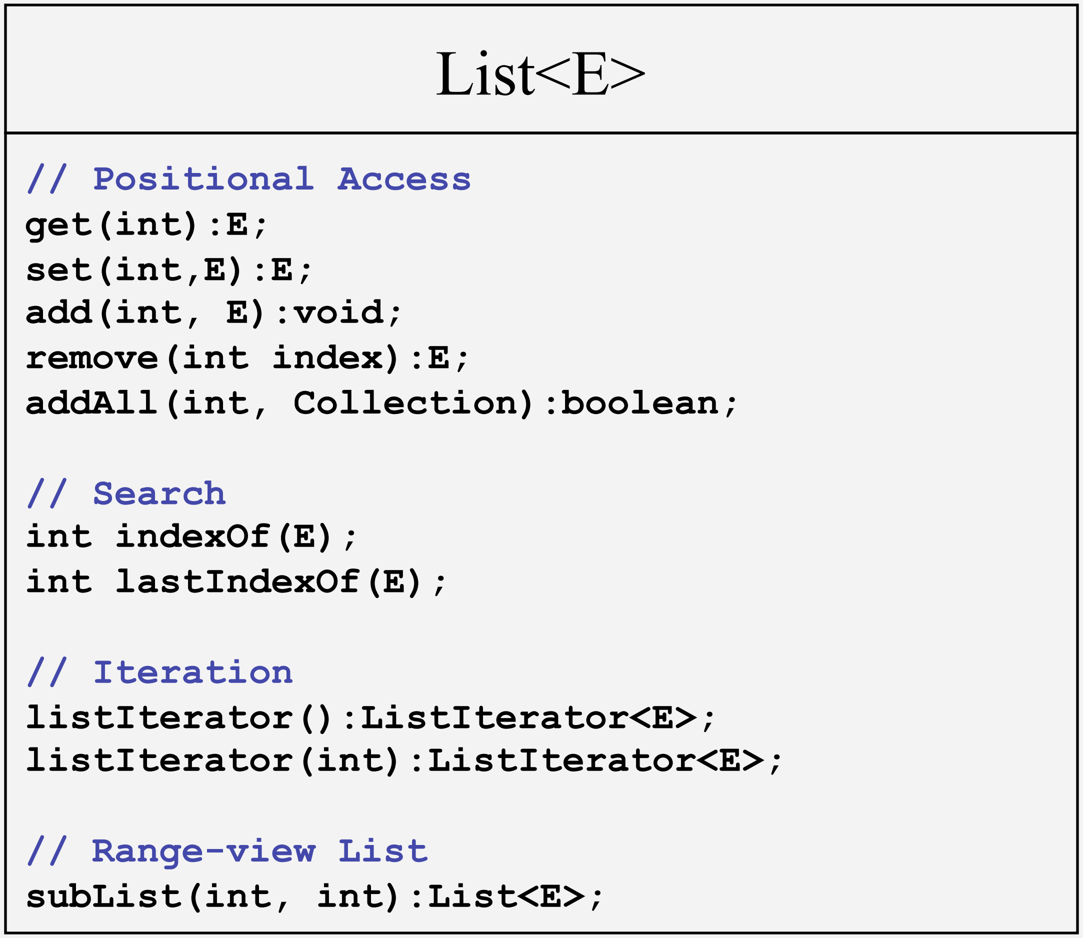
Instance of Iterator given by iterator() method in Collection

Operations

- Similar behaviour to operations used in *Scanner* to obtain a sequence of tokens
- Check if all elements have been visited (hasNext())
- Get next element in order imposed by the iterator (next())
- remove() the last element returned by next()

List Interface

(modified slide from Carl Reynolds)



List<E>

```
// Positional Access
get(int):E;
set(int,E):E;
add(int, E):void;
remove(int index):E;
addAll(int, Collection):boolean;

// Search
int indexOf(E);
int lastIndexOf(E);

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

// Range-view List
subList(int, int):List<E>;
```

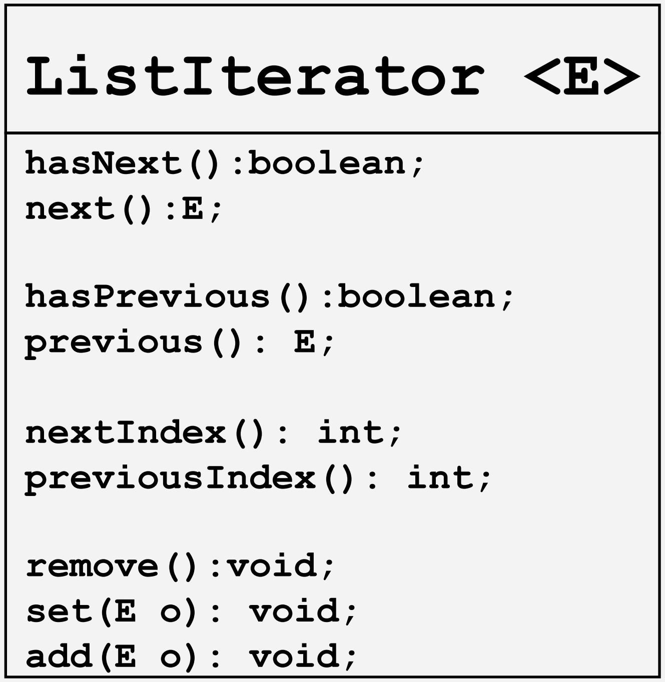
ListIterator

(modified slide from Carl Reynolds)

the `ListIterator` **interface extends** `Iterator`

Forward and reverse directions are possible

`ListIterator` **is available for Java Lists, such as the** `LinkedList` **implementation**



```
ListIterator <E>

hasNext():boolean;
next():E;

hasPrevious():boolean;
previous(): E;

nextIndex(): int;
previousIndex(): int;

remove():void;
set(E o): void;
add(E o): void;
```

The Collections Class

Operations for Manipulating Collections

Includes static operations for sorting, searching, replacing elements, finding max/min element, and to copy and alter collections in various ways.

(using this in lab5)

Note!

Collection is an interface for an abstract data type, *Collections* is a separate class for methods operating on collections.

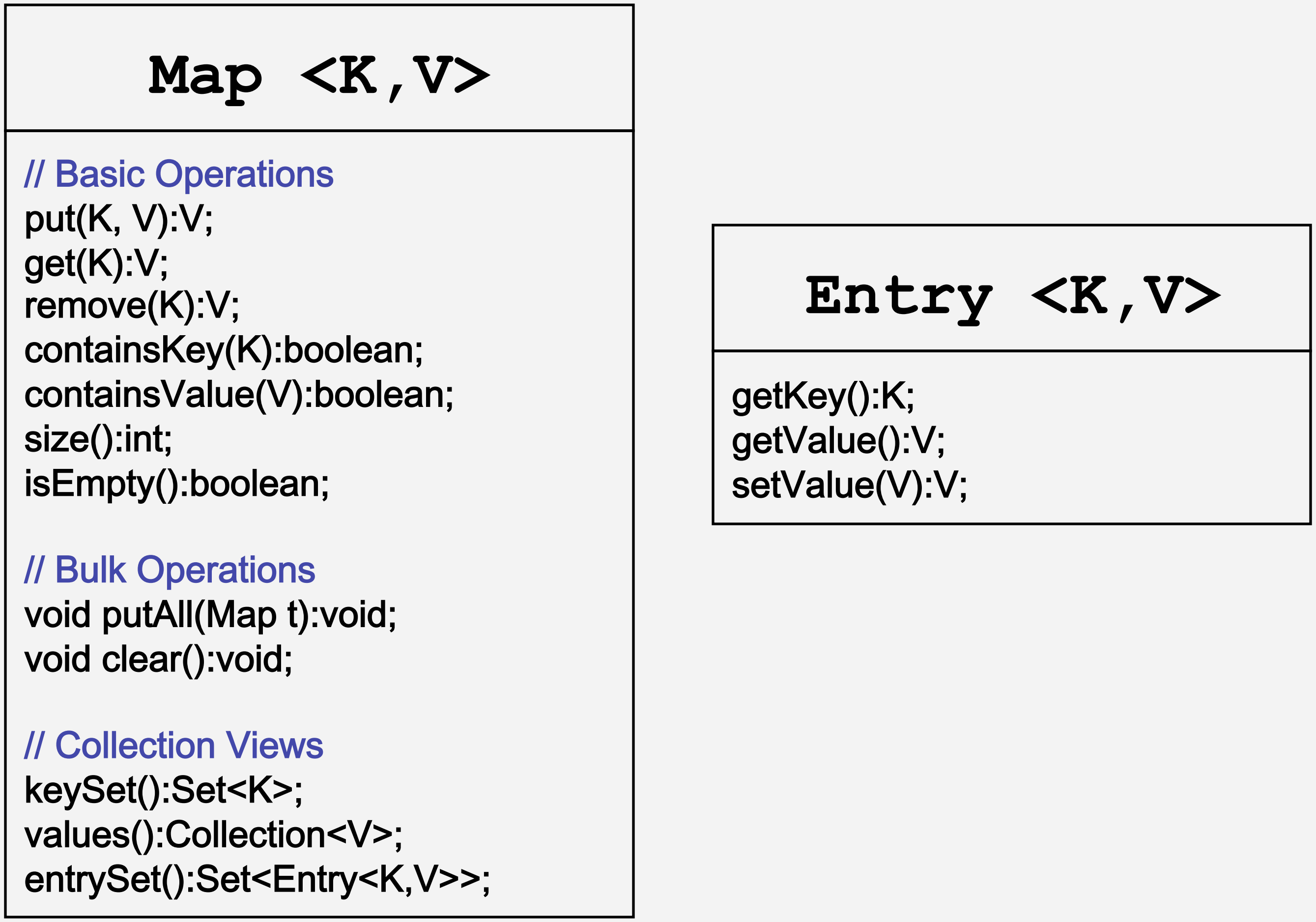
List: Example

TestArrayAndLinkedList.java

(course web page)

`Map <K,V>` Interface

(modified slide from Carl Reynolds)



`Map <K,V>`

// Basic Operations
put(K, V):V;
get(K):V;
remove(K):V;
containsKey(K):boolean;
containsValue(V):boolean;
size():int;
isEmpty():boolean;

// Bulk Operations
void putAll(Map t):void;
void clear():void;

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

`Entry <K,V>`

getKey():K;
getValue():V;
setValue(V):V;

Map Examples

CountOccurranceOfWords.java

(course web page)

TestMap.java (from text)

Comparator Interface (a generic class similar to *Comparable*)

(comparator slides adapted from Carl Reynolds)

You may define an alternate ordering for objects of a class using objects implementing the Comparator Interface (i.e. rather than using compareTo())

Sort people by age instead of name

Sort cars by year instead of Make and Model

Sort clients by city instead of name

Sort words alphabetically regardless of case

Comparator<T> Interface

One method:

```
compare( T o1, T o2 )
```

Returns:

```
negative if o1 < o2
Zero     if o1 == o2
positive if o1 > o2
```

Example Comparator: Compare 2 Strings regardless of case

```java
import java.util.*;
public class CaseInsensitiveComparator implements Comparator<String> {
  public int compare( String stringOne, String stringTwo ) {

    // Shift both strings to lower case, and then use the
    //  usual String instance method compareTo()
    return stringOne.toLowerCase().compareTo( stringTwo.toLowerCase() );
  }
}
```

Using a Comparator...

```
Collections.sort( myList, myComparator );
Collections.max( myCollection, myComparator );
Set myTree = new TreeSet<String>( myComparator );
Map myMap  = new TreeMap<String>( myComparator );
```

```
import java.util.*;
public class SortExample2B {
  public static void main( String args[] ) {

    List aList = new ArrayList<String>();

    for ( int i = 0; i < args.length; i++ ) {
        aList.add( args[ i ] );
    }
    Collections.sort( aList , new CaseInsensitiveComparator() );
    System.out.println( aList );
  }
}
```

...