

The Java Collection Framework: Interfaces, Classes, and Algorithms



1

CSD Univ. of Crete

Fall 2012

What is a Framework?

- "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes."
 - ◆ "Designing Reuseable Classes", Johnson and Foote 1988
- "A framework is a set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions."
 - ◆ "Building Object-Oriented Frameworks" Taligent/IBM



The Gang of Four on Frameworks

- "The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. A framework predefines these design parameters so that you, the application designer/implementer can concentrate on the specifics of your application. The framework captures the design decisions that are common to its application domain. Frameworks thus emphases design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately."
 - ◆ Gamma, Helm, Johnson, Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software" p. 25

3

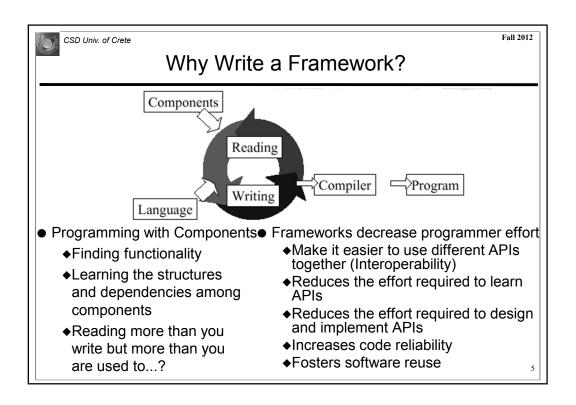


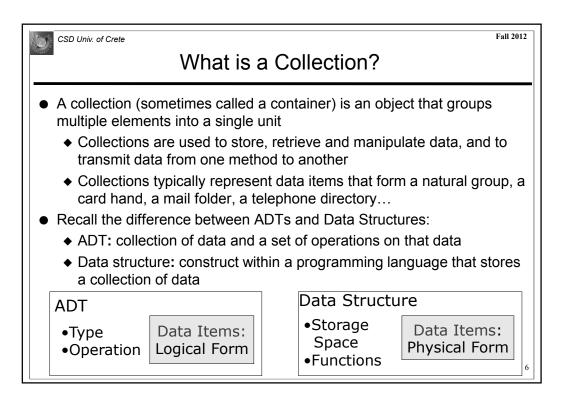
CSD Univ. of Crete

Fall 2012

More than just a Good O/O Implementation?

- "An abstract class is a design for a single object. A framework is the design of a set of objects that collaborate to carry out a set of responsibilities. Thus frameworks are larger scale designs than abstract classes. Frameworks are a way to reuse high-level design."
 - ◆"Reusing Object-Oriented Designs", Johnson and Russo, 1991
- "Frameworks are not simply collections of classes. Rather, frameworks come with rich functionality and strong wired-in interconnections between object classes that provide an infrastructure for the developer."
 - ◆Taligent, 1993







Given a Container, You ...

- Put an object in
- Take an object out
- Ask about a specific object
 - ◆Is it in the container?
 - ◆Is an equivalent object in the container?
 - ♦How many times?
- Retrieve an object by a key value
- Iterate over everything in the container
- Putting ADT theory into practice

7



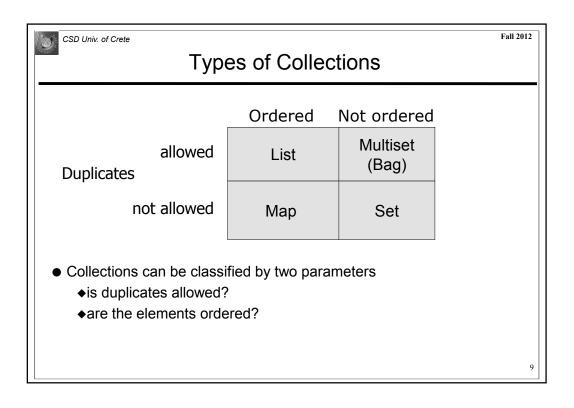
CSD Univ. of Crete

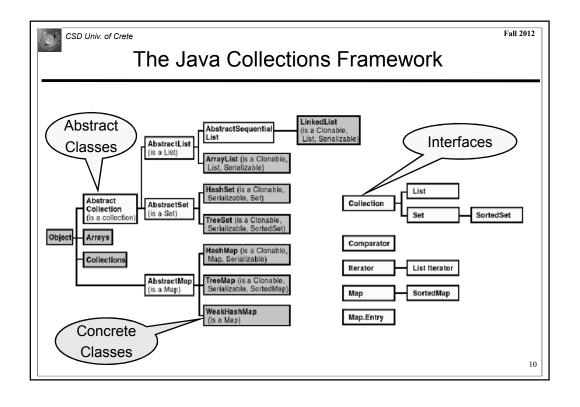
Fall 2012

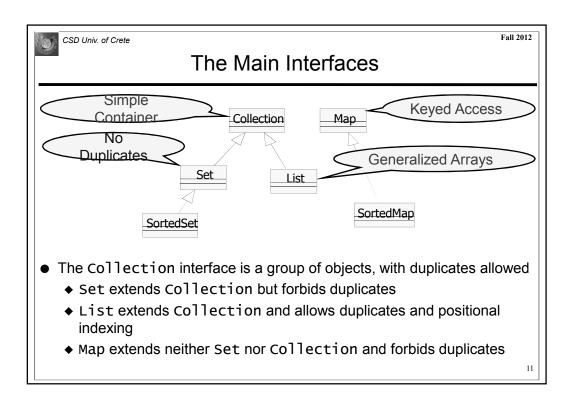
The Java Collections Framework

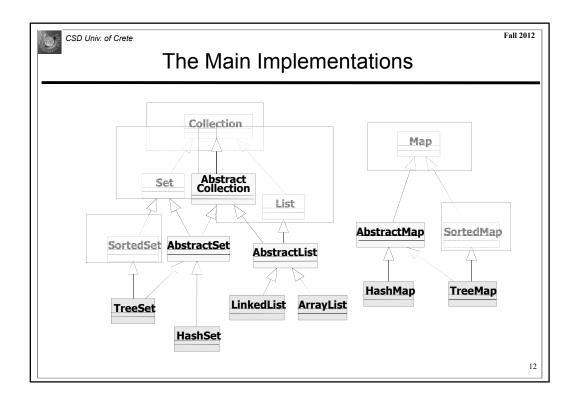
- A new framework in Java 2: Provide a basic set of "Object Containers"
 - ◆ Core data structures for everyday coding
- Predecessors:
 - ◆ C++'s Standard Template Library and ObjectSpace's JGL
 - ◆ JDK 1.02 : Vector, Hashtable, and Enumeration
- The Java Collections Framework provides:
 - ◆ Interfaces: 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
- Core Interfaces:
 - ◆ Collection
 - ◆ Set
 - ♦ List
 - ◆ Map
 - ◆ SortedSet
 - ◆ SortedMap

- Utility Interfaces
 - ◆ Comparator
 - ◆ Iterator
- Utility Classes
 - ◆ Collections
 - Arrays









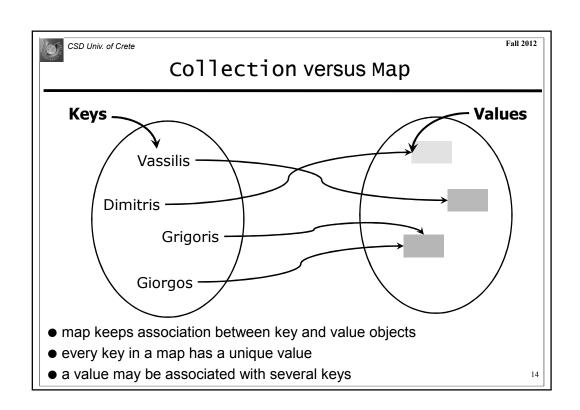


Collection versus Map

- A Collection is a group of objects
- Designed for access via Iterators
 - ◆You can add, remove, lookup
- Easily extended for sorting elements
- Examples:
 - ◆distinct words in a novel
 - ◆reserved keywords in java
 - ◆students taking HY252

- A Map (mathematical function) associates à value with each member key in its domain
 - ◆set of pairs (key-value), each pair representing a "one-directional mapping" from one set to another
- isolated items in the collection

 Designed for providing associative access to values stored by key
 - ◆The collection operations are available but they work with a keyvalue pair instead of an isolated
 - ◆Listing all (key,value) pairs requires using an iterator on the key set (domain of the map)
 - Examples
 - ◆A map of keys to database records
 - ◆A dictionary (words mapped to meanings)





Set versus List

- Sets are faithful to their mathematical definition
- No methods added to Collection but simply enforces "no duplicates" rule using the equals method
 - ◆More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element
- Access via Iterators
- Examples
 - ◆The set of uppercase letters 'A' through 'Z'
 - ◆The set of nonnegative integers { 0, 1, 2, ... }
 - ◆The empty set {}

- Lists are sequential structures
- Duplicates allowed
 - More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements
 - ◆It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare
- Implements get and remove methods using integer index

15



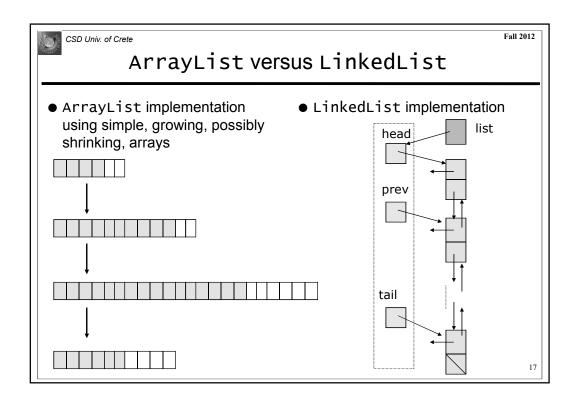
CSD Univ. of Crete

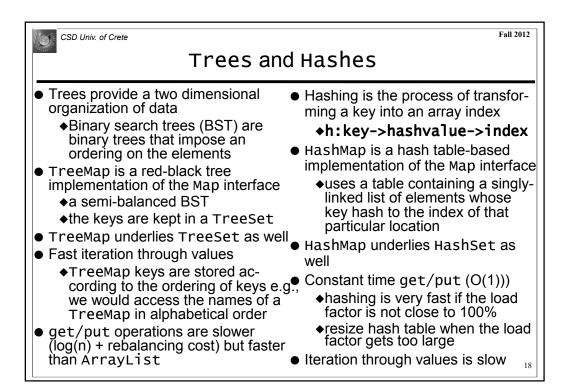
Fall 2012

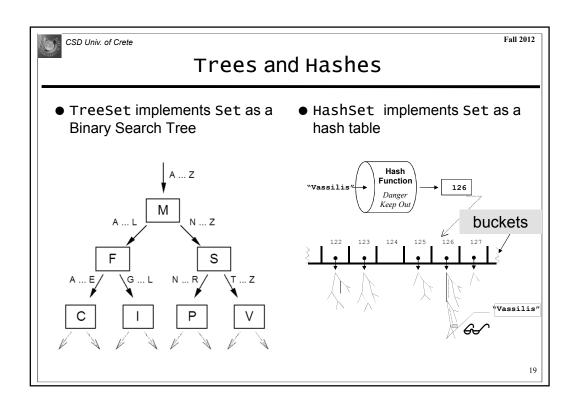
ArrayList versus LinkedList

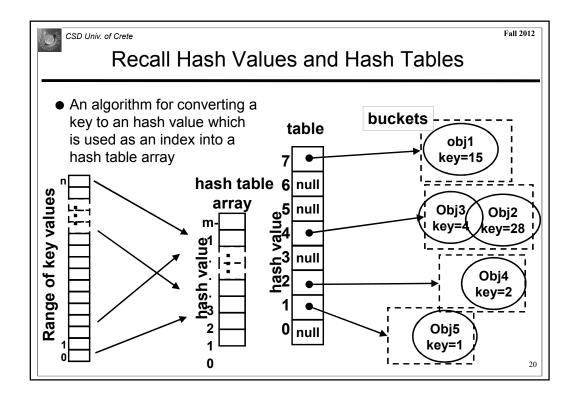
- ArrayList is a wrapper around an array (resizes when it gets full)
 - ◆Implements all optional list operations, and permits all elements, including null
 - ◆This class provides methods to manipulate the size of the array that is used to store the list
- Insertion to the front or middle is expensive (O(n))
- Provides fast iterator and get(int) methods

- LinkedList is a classic doubly linked list
 - ◆Implements all optional list operations, and permits all elements, including null
 - ◆Provides uniformly named methods to get, remove and insert an element at the beginning and end of the list (i.e., they can be used as a stack, queue, or double-ended queue (deque))
- Constant time insertion and removal (O(1)) anywhere in the list
- Slower iterator
- get(int) very slow
 - ◆implemented by iterating









Fall 2012 CSD Univ. of Crete The Collection Interface Found in the java.util package // Basic Operations size():int; Optional methods throw isEmpty():boolean; UnsupportedOperation contains(Object):boolean; **Exception** if the // Optional add(Object):boolean; implementing class does remove(Object):boolean; // Optional not support the operation iterator():Iterator; // Bulk Operations Bulk operations perform containsAll(Collection):boolean; some operation on an addAll(Collection):boolean; // Optional entire Collection in a single removeAll(Collection):boolean;// Optional shot retainAll(Collecton):boolean; // Optional // Optional clear():void; The **toArray** methods allow the contents of a // Array Operations Collection to be toArray():Object[]; translated into an array toArray(Object[]):Object[];



CSD Univ. of Crete

Fall 2012

The Collection Interface

boolean **add**(Object o): Ensures that this collection contains the specified element

boolean **addAll**(Collection c): Adds all of the elements in the specified collection to this collection

void clear(): Removes all of the elements from this collection

boolean **contains**(Object o): Returns true if this collection contains the specified element

boolean **containsAll**(Collection c): Returns true if this collection contains all of the elements in the specified collection

boolean **equals**(Object o): Compares the specified object with this collection for equality

int **hashCode()**: Returns the hash code value for this collection

boolean isEmpty(): Returns true if this collection contains no elements,

The Collection Interface

- Iterator **iterator**(): Returns an iterator over the elements in this collection
- boolean **remove**(Object o): Removes a single instance of the specified element from this collection, if it is present
- boolean **removeAll**(Collection c): Removes all this collection's elements that are also contained in the specified collection
- boolean **retainAll**(Collection c): Retains only the elements in this collection that are contained in the specified collection
- int **size**(): Returns the number of elements in this collection
- Object[] toArray(): Returns an array containing all of the elements in this collection
- Object[] toArray(Object[] a): Returns an array containing all of the elements in this collection whose runtime type is that of the specified array
- Examples:
- `Object[] a= c.toArray();

String[] a= (String[]) c.toArray(new String[0]);

23

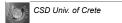


CSD Univ. of Crete

Fall 2012

Creating and Using a Collection

- Collection is actually an interface
 - ◆Each kind of Collection has one or more implementations
 - ♦You can create new kinds of Collections
 - ♦When you implement an interface, you supply all the required methods
- All Collection implementations should have two constructors:
 - ◆A no-argument constructor to create an empty Collection
 - ◆A constructor with another Collection as argument
- All the standard implementations obey this rule, but If you implement your own Collection type, this rule cannot be enforced, because an Interface cannot specify constructors
- Note that most methods e.g. boolean containsAll(Collection c); are defined for any type of Collection, and take any type of Collection as an argument
 - ◆This makes it very easy to work with different types of Collections



The Set Interface

- A **Set** is a **Collection** that cannot contain duplicate elements
 - ◆ Set models the mathematical set abstraction
- The **Set** interface extends **Collection** and contains no methods other than those inherited from **Collection**
 - ◆ It adds the restriction that duplicate elements are prohibited
 - ◆ Two **Set** objects are equal if they contain the same elements
- The bulk operations perform standard set-algebraic operations: Suppose s1 and s2 are **Set**s
 - ◆ s1.containsAll(s2): Returns true if s2 is a subset of s1
 - ◆ s1.addAll(s2): Transforms s1 into the union of s1 and s2 (The union of two sets is the set containing all the elements contained in either set)
 - ◆ s1.retainAll(s2): Transforms s1 into the intersection of s1 and s2 (The intersection of two sets is the set containing only the elements that are common in both sets)

25

Fall 2012 CSD Univ. of Crete **Set Operations** s.addAll(t)union $s \leftarrow s \cup t$ intersection $s \leftarrow s \cap t$ s.retainAll(t)difference $s \leftarrow s \setminus t$ s.removeAll(t)inclusion $s \supset t$ s.containsAll(t) $s \leftarrow s \cup \{o\}$ insert s.add(o) $s \leftarrow s \setminus \{o\}$ delete s.remove(o)membership $o \in s$ s.contains(o)cardinality s.size()|s|emptiness $s = \emptyset$ s.isEmpty()make empty $s \leftarrow \emptyset$ s.clear()iteration s.iterator()

CSD Univ. of Crete Fall 2012

The List Interface

- A List is an ordered Collection (sometimes called a sequence)
 - ◆ In a sequence every object (except the first and last) has a predecessor and a successor, while occupies a unique position in the sequence
 - ◆ Lists may contain duplicate elements
- Inherits operations from **Collection** as:
 - ◆ add(Object) //Append Object to receiver (add to the end of the sequence)
 - ◆ contains(Object) // Returns TRUE if the receiver contains the Object
 - ◆ remove(Object) //Remove the first occurrence of Object (adjust rest of the list)
- The **List** interface includes additional operations for:
 - Positional Access
 - ◆ Search
 - ◆ List Iteration
 - ◆ Range-view
- Think of the Vector class

27

Fall 2012 CSD Univ. of Crete The List Interface // Positional Access //Returns a copy of the Object located at the index Object get(int): //Overwrite contents of list at position of index with Object object set(int,Object); // Optional //Insert Object at indicated index (adjust the rest list) void add(int, Object); // Optional //Remove Object at indicated index (adjust the rest list)
Object remove(int index); // Optional
boolean addAll(int, Collection);// Optional ' Search //Returns the index of the first occurrence of Object - //returns length of list if the Object is not present int indexOf(Object);
int lastIndexOf(Object); ' Iteration //Returns a ListIterator to traverse the referenced container
ListIterator listIterator();
ListIterator listIterator(int); Range-view List List subList(int, int);

CSD Univ. of Crete

The Map Interface

- Replaces java.util.Dictionary interface
 - ◆An object that maps keys to values
 - ◆Each key can have at most one value
- Ordering may be provided by implementation class, but not guaranteed
- Methods
 - ◆Set keySet();
 - *Collection values();
 - ◆Set entrySet(); // returns a set view of the mappings
- Map.Entry
 - ◆Object that contains a key-value pair
 - getKey(), getValue()
- Thread safety
 - ◆The collections returned are backed by the map
 - When the map changes, the collection changes
 - ◆Behavior can easily become undefined
 - Be very careful and read the docs closely

Fall 2012

```
Fall 2012
  CSD Univ. of Crete
                   The Map Interface
// Basic Operations
Object put(Object, Object); //If the map already contains a
// given key it replaces the value associated with that key
Object get(Object); // returns the corresponding value
Object remove(Object); // removes the pair with that key
boolean containsKey(Object);
boolean containsValue(Object);
int size();
boolean isEmpty();
// Bulk Operations
void putAll(Map t); //copies one Map into another
void clear();
// Collection Views
Set keySet();
Collection values();
Set entrySet();//returns a set of Map.Entry (key-value) pairs
```

CSD Univ. of Crete

Fall 2012

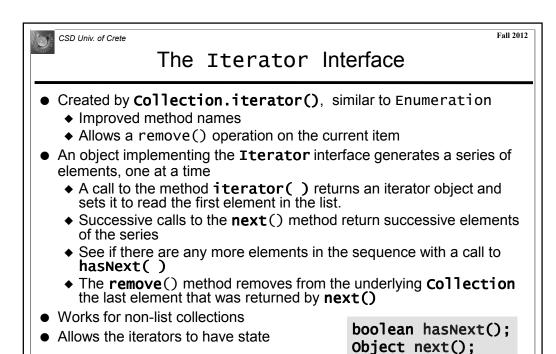
The Map. Entry Interface

This is a small interface for working with the Collection returned by entrySet()

```
public interface Entry {
  Object getKey();
  Object getValue();
  Object setValue(Object);
}
```

 Can get elements only from the **Iterator**, and they are only valid during the iteration

```
Fall 2012
CSD Univ. of Crete
                          Map Operations
                    f \leftarrow f|_{dom(f)\setminus\{k\}}f \leftarrow [k \mapsto o] \cup f|_{dom(f)\setminus\{k\}}
 restriction
                                                         f.remove(k)
 extension
                                                         f.\mathtt{put}(k,o)
 extension
                     f \leftarrow g \cup f|_{dom(f) \setminus dom(g)}
                                                         f.\mathtt{putAll}(g)
                                 f \leftarrow []
make empty
                                                         f.clear()
   lookup
                                  f(k)
                                                         f.\mathtt{get}(k)
                             k \in dom(f)
   defined
                                                         f.\mathtt{containsKey}(k)
     hits
                             \exists k. f(k) = o
                                                         f.containsValue(o)
                               |dom(f)|
     size
                                                         f.size()
                             dom(f) = \emptyset
 emptiness
                                                         f.isEmpty()
   domain
                                dom(f)
                                                         f.\texttt{keySet}()
 co-domain
                         [f(k)|k \in dom(f)]
                                                         f.values()
                     \{(k, f(k))|k \in dom(f)\}
                                                         f.entrySet()
 map pairs
```



void remove();

```
Fall 2012
  CSD Univ. of Crete
                     Using Iterators

    Replace

      int counter;
      int size = collection.size();
      for (counter=0; counter<size; counter++) {</pre>
         Object value = collection.get(counter);
• with:
             Iterator i = collection.iterator();
             while(i.hasNext()) {
              Object value = i.next();
                  for (Object o: collection) {
or even
                     System.out.println(o);
                  }
```

```
Fall 2012
  CSD Univ. of Crete
                        Using Iterators
• With an iterator one can write generic methods to handle any collection
public class Printer {
                                                The method receives an
  static void printAll (Iterator e) {
                                                iterator object as its
                                                argument with no
     while (e.hasNext( ) )
                                                distinction made about
        System.out.print(e.next( )+" ");
                                                what collection produced it
     System.out.println();
}
public class FrogPond {
public static void main(String [ ] args)
     ArrayList v = new ArrayList();
                                                  Obtain an iterator from
     for (int i = 0; i < args[0]; i++)
                                                  the ArrayList and pass
           v.add(new Frog(i));
                                                  it to the Printer
     Printer.printAll(v.iterator()); __
  }
}
```

```
Removing items via an Iterator

• Iterators can also be used to modify a Collection:

Iterator i = collection.iterator();
while (i.hasNext())
{
    if (object.equals(i.next()))
    {
        i.remove();
    }
}
```

CSD Univ. of Crete

Fall 2012

The ListIterator Interface

- A ListIterator is produced by any Collection implementing the List interface
 - ◆interface ListIterator extends Iterator
 - Created by List.listIterator()
- It has an additional constructor that takes an index value as a parameter and returns an iterator set to begin visiting the element at this location on the first call to next()

*ListIterator(int);



- Adds methods to
 - ◆traverse the List in either direction
 - ◆modify the List during iteration
- A client program can maintain multiple iterators to facilitate searches and swaps inside of a list

31

CSD Univ. of Crete

Fall 2012

The ListIterator Interface

 In addition to methods next(), hasNext(), and remove() that are inherited from interface Iterator, ListIterator provides the following additional methods

public int nextIndex(); //returns index of the element that
would be called by next()

public int previousIndex(); //returns index of the element
that would be called by previous()

public boolean hasPrevious();

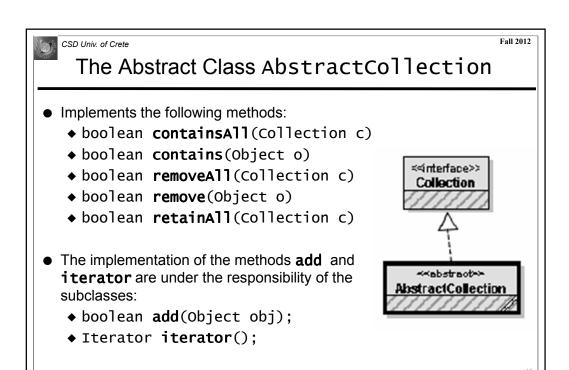
public Object previous() throws NoSuchElementException;

public void add(Object o) throws
UnsupportedOperationException, ClassCastException,
IllegalArgumentException; //inserts the object into the list
immediately before the current index position

public void set(Object o) throws
UnsupportedOperationException, ClassCastException,
IllegalArgumentException; //overwrites the last element
returned by next or previous with the specified element

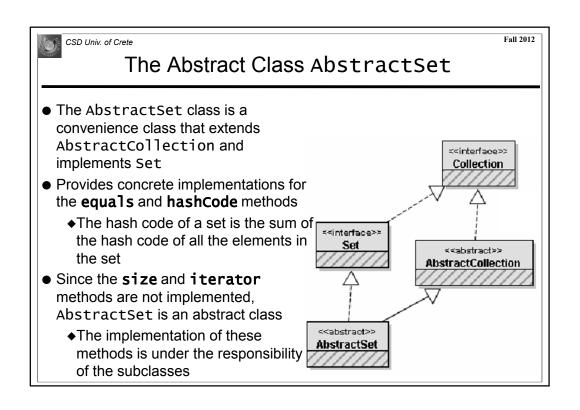
```
Using a ListIterator

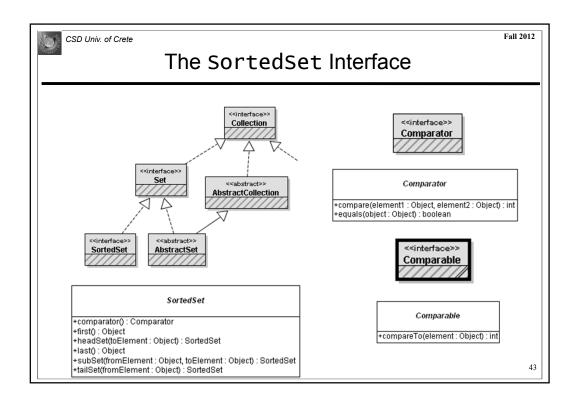
List list = new LinkedList();
//fill the list with objects ...
//Produces a ListIterator set to visit the element
//at index 5
ListIterator itr = list.getListIterator(5);
while (itr.hasNext()) {
    if (object.compareTo(itr.next()) < 0) {
        break;
    }
}
//A ListIterator allows items to be inserted in a List
itr.add(object);
```

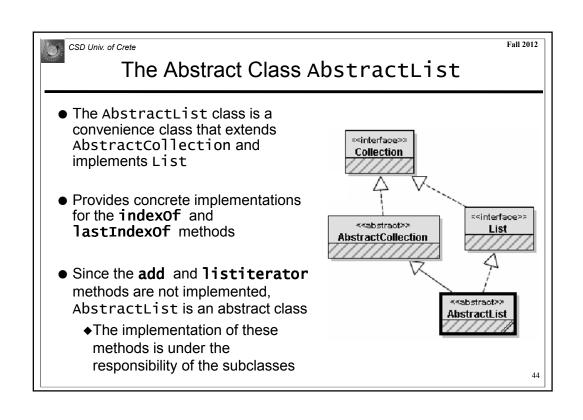


```
AbstractCollection

public boolean
   retainAll(Collection c){
   boolean modified = false;
   Iterator e = iterator();
   while (e.hasNext()) {
      if(!c.contains(e.next())) {
        e.remove();
      modified = true;
      }
   }
   return modified;
}
```





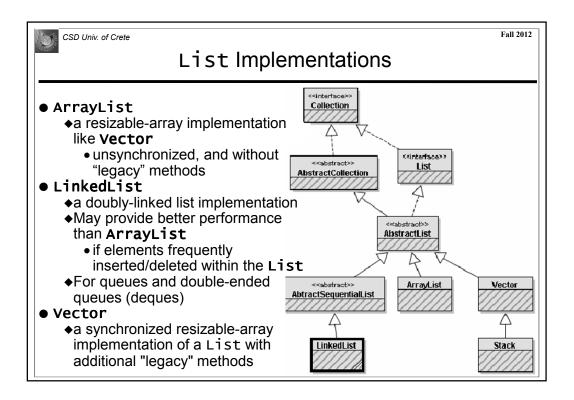


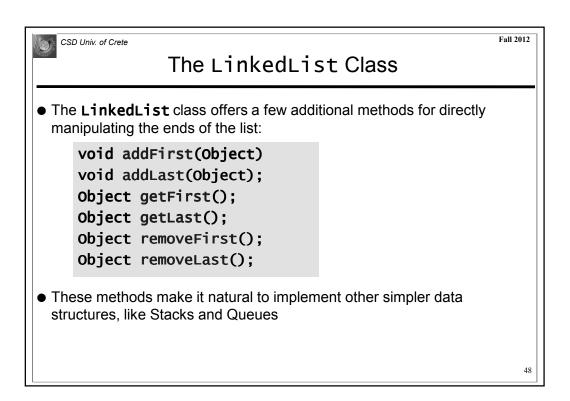
CSD Univ. of Crete		
- Salaria	Implementation Classes	

Interface	Implementation				Historical	
	Hash table	Resizable array	Tree (sorted)	Linked list		
Set	HashSet		TreeSet			
List		ArrayList		LinkedList	Vector Stack	
Мар	наѕһмар		ТгееМар		HashTable Properties	

- ■When writing programs think about interfaces and not implementations
 ■This way the program does not become dependent on any added methods in a given implementation, leaving the programmer with the freedom to change implementations
- An implementation class may elect not to support a particular method of the interface
 - ●UnsupportedOperationException is a runtime (unchecked) exception

Fall 2012 CSD Univ. of Crete Set Implementations • HashSet ≺interface+ Collection ◆a **Set** backed by a hash table TreeSet ≪interface>> ◆A semi-balanced binary tree Set **AbstractCollection** implementation ◆Imposes an ordering on its elements ≺≺interface≫ SortedSet TreeSet HashSet 46









Using LinkedLists

- A few things to be aware of:
 - ◆it is really bad to use the positional indexing features copiously of LinkedList if you care at all about performance
 - This is because the **LinkedList** has no memory and must always traverse the chain from the beginning
 - ◆Elements can be changed both with the List and ListIterator objects
 - That latter is often more convenient
 - ◆You can create havoc by creating several iterators that you use to mutate the List
 - There is some protection built-in, but best is to have only one iterator that will actually mutate the list structure

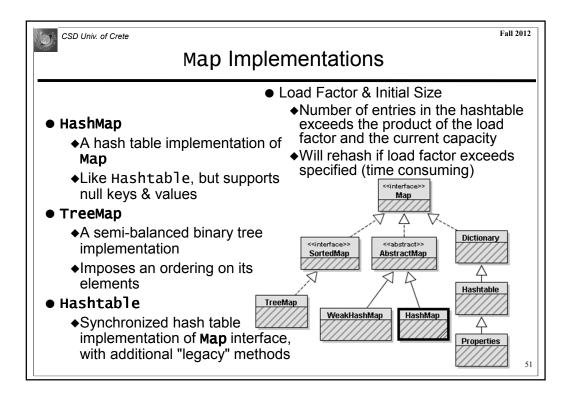


CSD Univ. of Crete

Fall 2012

The ArrayList Class

- Additional methods for managing size of underlying array
 - ◆void **ensureCapacity**(int minCapacity): Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument
 - ◆void trimToSize(): Trims the capacity of this ArrayList instance to be the list's current size
- methods size(), isEmpty(), get(), set(), iterator() and listIterator() all run in constant time
- Adding n elements take O[n] time
- Can explicitly grow capacity in anticipation of adding many elements
- Note: legacy Vector class almost identical
 - ◆Main differences are naming and synchronization





CSD Univ. of Crete

Fall 2012

Load Factor

- The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased
 - ♦When the number of entries in the hashtable exceeds the product of the load factor and the current capacity, the capacity is increased by calling the rehash method
- Generally, the default load factor (.75) offers a good tradeoff between time and space costs
 - ◆Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most Hashtable operations, including get and put)
- The initial capacity controls a tradeoff between wasted space and the need for rehash operations, which are time-consuming
 - ◆No rehash operations will *ever* occur if the initial capacity is greater than the maximum number of entries the Hashtable will contain divided by its load factor. However, setting the initial capacity too high can waste space.
- If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.



The Hashtable Class

- Like a **HashMap** but older version
- Constructors:

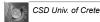
Hashtable() //Constructs a new, empty hashtable with a default initial capacity (11) and load factor, which is 0.75.

Hashtable(int initialCapacity) //Constructs a new, empty
hashtable with the specified initial capacity and
default load factor, which is 0.75

Hashtable(int initialCapacity, float loadFactor)
//Constructs a new, empty hashtable with the specified
initial capacity and the specified load factor

Hashtable(Map t) //Constructs a new hashtable with the same mappings as the given Map

53

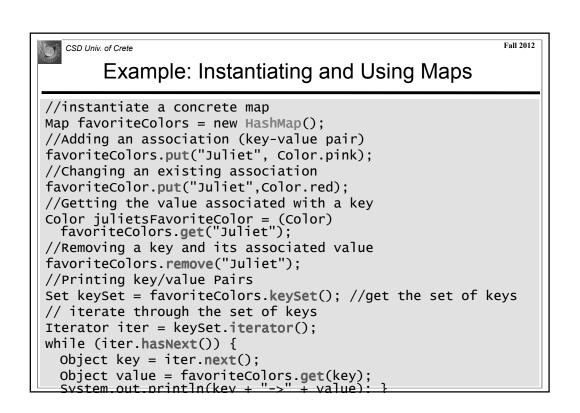


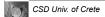
Fall 2012

Other J2SE Implementations

- Legacy (since 1.0)
 - ◆java.util.Vector
 - ◆iava.util.Stack
 - ◆java.util.Hashtable
 - ♦java.util.Properties
- J2SE 1.2
 - ◆java.util.WeakHashMap
- J2SE 1.4
 - ◆java.util.LinkedHashSet
 - ◆java.util.LinkedHashMap
 - ◆java.util.IdentityHashMap
- J2SE 1.5
 - ◆java.util.EnumSet
 - ◆java.util.EnumMap
 - ◆java.util.PriorityQueue
 - ◆java.util.concurrent.*

```
Fall 2012
 CSD Univ. of Crete
      Example: Instantiating and Using Sets
public class TestCollection {
  public static void main(String args[]) {
   // Create an empty set
   Collection set = new HashSet();
   // Populate the set
   set.add(new Integer(47));
                                                   47
   set.add(new Double(3.14));
   set.add(new Character('h'));
                                                  3.14
   // iterate through the set of keys
   Iterator iter = set.iterator();
                                                    h
   while (iter.hasnext()) {
   //Assume items are printed in same order
   //they were put in
  System.out.println(iter.next());
   } // end while
  } // end main
} // end TestCollection
```





Hash based Implementation

- Hash based implementation stores set elements (map keys) using a hash function
- A suitable hash function is defined in class Object (method hashCode) and inherited by all subclasses
- If a class overrides the equals method defined in class Object it is also necessary to override the hashCode method to make HashSet and HashMap work correctly

57



CSD Univ. of Crete

Fall 2012

Tree based Implementation

• TreeSet

- ◆iteration gives elements in sorted order
- ◆To use a TreeSet.
 - either your objects must implement interface Comparable
 - or you must provide a **Comparator** object

TreeMap

- ◆the keys are kept in a TreeSet
- ◆To use a **TreeMap**
 - either your keys must implement interface Comparable
 - or you must provide a **Comparator** object for the keys
 - there is no requirement for the values

CSD Univ. of Crete

Fall 2012

What About User Objects?

- The Collections framework will work with any Java class
- You need to be sure you have defined
 - ◆equals()
 - ◆hashCode()
 - ◆compareTo() or compare()
- Don't use mutable objects for keys in a Map
- The Map hashCode() returns distinct integers for distinct objects
 - ◆If two objects are equal according to the equals() method, then the hashCode() method on each of the two objects must produce the same integer result
 - ◆When hashCode() is invoked on the same object more than once, it must return the same integer, provided no information used in equals comparisons has been modified
 - ♦It is not required that if two objects are unequal according to equals() that hashCode() must return distinct integer values

59



CSD Univ. of Crete

Fall 2012

Sorting and Comparing

- Comparable interface
 - ◆Must be implemented by all elements in SortedSet
 - ◆Must be implemented by all keys in SortedMap
 - ◆Method: int compareTo(Object o)
 - ◆Defines "natural order" for that object class
- Comparator interface
 - ◆Defines a function that compares two objects
 - ◆Can design custom ordering scheme
 - ◆Method: int compare(Object o1, Object o2)
- Total vs. Partial Ordering
 - ◆Technical, changes behavior per object class



Return Abstract Collections

- OO is based on the notion of encapsulation
 - hiding implementation details behind interfaces
- Therefore: Return the most abstract interface that can possibly work
 - ◆Return a List instead of an ArrayList
 - ◆Return a Collection instead of a List
 - ◆Return an Iterator instead of a Collection

61



The Collections Utility Class

- This class consists exclusively of static methods that operate on or return Collections and Maps
 - Polymorphic algorithms that operate on collections
 - Wrappers which return a new collection backed by a specified collection
 - Plus a few other odds and ends for creating synchronized collection classes, and for creating read-only collection classes
- Most of the algorithms operate on List objects, but a couple of them (max and min) operate on arbitrary Collection objects

+binarySearch(list: List, key: Object): int
+binarySearch(list: List, key: Object, c: Comparator): int
+binarySearch(list: List, key: Object, c: Comparator): int
+copy(src: List, des: List): void
+enumeration(c: final Collection): Enumeration
+fill(list: List, o: Object): void
+max(c: Collection): Object
+max(c: Collection): Object
+min(c: Collection): Object
+min(c: Collection): Object
+min(c: Collection): Object
+reverseOrder(): Comparator): Object
+reverseOrder(): Comparator
+shuffle(list: List): void
+shuffle(list: List): void
+shuffle(list: List): void
+shuffle(list: List): roid
+shuffle(list: List): void
+shuffle(list: List): void
+singleton(c: Object): List
+singletonMap(key: Object, value: Object): Map
+sort(list: List): void
+sort(list: List): c: Comparator): void
+synchronizedCollection(c: Collection): Collection
+synchronizedList(list: List): List
+synchronizedSortedMap(s: SortedMap): SortedMap
+synchronizedSortedMap(s: SortedMap): SortedMap
+synchronizedSortedMap(s: SortedSet): SortedSet
+unmodifiedCollection(c: Collection): Collection
+unmodifiedMap(m: Map): Map
+unmodifiedSortedMap(s: SortedMap): SortedMap
+unmodifiedSortedMap(s: SortedMap): SortedMap
+unmodifiedSortedMap(s: SortedMap): SortedMap
+unmodifiedSortedMap(s: SortedMap): SortedMap

Collections

```
Fall 2012
   CSD Univ. of Crete
         Static Method Collections.sort()

    The sort operation uses a slightly

  optimized merge sort algorithm
                                     public static void sort(List
   ◆Fast: This algorithm is guaranteed
                                        list, Comparator c){
     to run in n log(n) time, and runs
                                      Object a[] = list.toArray();
     substantially faster on nearly
                                       Arrays.sort(a, c);
     sorted lists
                                       ListIterator i =
   ◆Stable: That is to say, it doesn't
                                        list.listIterator();
     reorder equal elements
                                       for (int j=0;j<a.length;</pre>
• SortedSet, SortedMap interfaces
                                        j++) {
   ◆Collections that keep their
                                            i.next();
     elements sorted
                                           i.set(a[j]);

    Iterators are guaranteed to

                                        }
     traverse in sorted order (only
     SortedSet)

    Ordered Collection Implementations

   ◆TreeSet, TreeMap
```

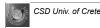
```
import java.util.*;

public class SortExample {
    public static void main( String args[] ) {
        List l = new ArrayList();

        for ( int i = 0; i < args.length; i++ )
            l.add( args[ i ] );

        Collections.sort( l );

        System.out.println( l );
    }
}</pre>
```



Other Algorithms

- Other algorithms provided by the Collections class include
 - Singleton
 - Collections.singleton(e) returns an immutable set containing only the element e
 - c.removeAll(Collections.singleton(e)); will remove all occurrences of e from the Collection c
 - Shuffling
 - Data manipulation
 - reverse()
 - fill()
 - copy()
 - Searching
 - Finding extreme values
 - max()
 - min()



CSD Univ. of Crete

Fall 2012

The Arrays Utility Class

- It is too bad that arrays are not collections
 - ◆You loose all of the power provided by the collection framework
- The class Arrays contains
 - various static methods for manipulating arrays (such as sorting, searching, and comparing arrays, as well as filling array elements)
 - It also contains a method for converting arrays to lists
- Object-based Array sorting
 - Arrays.sort(Object[])
 - ◆Equivalent methods for all primitive types (e.g., int[])

Arrays

+asList(a: Object[]): List +binarySearch(a: byte[],key: byte); int +binarySearch(a: char[], key: char): int +binarySearch(a: double[], key: double]; int +binarySearch(a: float[], key: int): int +binarySearch(a: int[], key: int): int +binarySearch(a: long[], key: long): int +binarySearch(a: Object[], key: Object): int +binarySearch(a: Object[], key: Object, c: Comparator): int +binarySearch(a: short[], key: short): int +binarySearch(a: short[], key: Short): int +cyquals(a: boolean[], a2: boolean[]): boolean

+binarySearch(a: short[], key: short): int
+equals(a: boolean[], a2: boolean[]): boolean
+equals(a: byte[], a2: byte[]): boolean
+equals(a: char[], a2: char[]): boolean
+equals(a: double[], a2: double[]): boolean
+equals(a: float[], a2: float[]): boolean
+equals(a: int[], a2: int[]): boolean
+equals(a: int[], a2: long[]): boolean
+equals(a: long[], a2: long[]): boolean
+equals(a: boolean[], a2: boolean
+equals(a: boolean[], a2: boolean
+equals(a: boolean[], a2: boolean
+fill(a: boolean[], rand boolean boolean
+fill(a: boolean[], rand boolean bo

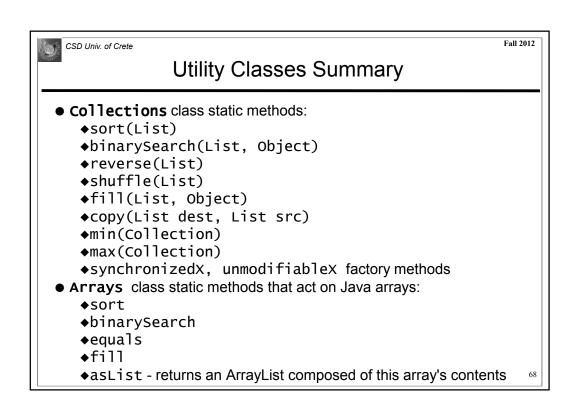
Overloaded fill method for char, byte, short, int, long, float, double,

sort(a: byte[]) : void

sort(a: byte[], fromIndex: int, toIndex: int) : void

Overloaded sort method for char, short, int, long, float, double, and

```
import java.util.*;
public class SortExample {
    public static void main( String args[] ) {
        Arrays.sort( args );
        List l = Arrays.asList( args );
        System.out.println( l );
    }
}
```



Fall 2012 CSD Univ. of Crete **Example: Counting UniqueWords** import java.io.*; import java.util.*; public class UniqueWords { public static void main(String args[]) { // Usage check & open file if (args.length != 1) { System.err.println("Usage: java UniqueWords word-file"); System.exit(1); } StreamTokenizer in = null; try { in = new StreamTokenizer(new BufferedReader (new FileReader (args[0])); in.ordinaryChar('.'); } catch (FileNotFoundException e) { System.err.println("UniqueWords: " + e.getMessage()); System.exit(1);}

```
Fall 2012
CSD Univ. of Crete
        Example: Counting UniqueWords
    try {
                                        Create Empty Set -
        Set set = new HashSet();
        while ( ( in.nextToken() != in.TT_EOF ) ) {
                 if ( in.ttype == in.TT_WORD )
                     set.add( in.sval );
        System.out.println("There are " + set.size() +
                            " unique words" );
        System.out.println( set );
    catch ( IOException e ) {
        System.err.println("Uniquewords: " +
                            e.getMessage());
        System.exit( 1 );
```

```
Fall 2012
CSD Univ. of Crete
           You Want Them Sorted?
   try {
       SortedSet set = new TreeSet();
       while ( ( in.nextToken() != in.TT_EOF ) ) {
         if ( in.ttype == in.TT_WORD )
             set.add( in.sval );
         }
       System.out.println( set );
    catch ( IOException e ) {
       System.err.println("UniqueWords: " +
                         e.getMessage() );
       System.exit( 1 );
     }
   }
```

```
Fall 2012
CSD Univ. of Crete
                        Or
    try {
       Set set = new HashSet();
       while ( ( in.nextToken() != in.TT_EOF ) ) {
         if ( in.ttype == in.TT_WORD )
              set.add( in.sval );
       System.out.println(new TreeSet(set) );
                                          Copy Set
    catch ( IOException e ) {
       System.err.println("UniqueWords: " +
                         e.getMessage() );
       System.exit( 1 );
     }
   }
```

```
Fall 2012
 CSD Univ. of Crete
                   Pretty Output is Good
try {
    SortedSet set = new TreeSet();
    while ( ( in.nextToken() != in.TT_EOF ) ) {
      if ( in.ttype == in.TT_WORD )
           set.add( in.sval );
    System.out.println("There are " + set.size() + " unique words" );
    Iterator elements = set iterator();
    System.out.println();
    while ( elements.hasNext() )
            System.out.println( elements.next() );
catch ( IOException e ) {
    System.err.println( "UniqueWords: " + e.getMessage() );
    System.exit( 1 );
}
```

```
Example: Counting UniqueWords

try {
    Map map = new HashMap();
    Integer one = new Integer(1);

while ((in.nextToken()!=in.TT_EOF)) {
    if (in.ttype == in.TT_WORD) {
        Integer freq = (Integer) map.get(in.sval);

    if (freq == null)
        freq = one;
    else
        freq = new Integer(freq.intValue() + 1);

    map.put(in.sval, freq);
    }
}
```



Example: Counting UniqueWords

CSD Univ. of Crete

Fall 2012

Dealing with Changes of Collection Objects

- Modifiable / Unmodifiable
 - ◆ Modifiable: Collections that support modification, "transformer" operations, e.g., add(), remove(), clear()
 - ◆Unmodifiable: Collections that do not support any modification operations
- Mutable / Immutable
 - ◆Immutable: Collections that guarantee that no change in the Collection will ever be observable via "selector" operations, e.g., such as iterator(), size(), contains()
 - ◆Mutable: Collections that are not immutable
- Fixed-size / Variable-size
 - ◆Fixed-size: Lists that guarantee that their size will remain constant even though the elements may change
 - ◆ Variable-size: Lists that are not fixed-size are referred to as

CSD Univ. of Crete Fall 2012



Wrapper Implementations

- Basic Containers are sometimes inadequate
 - ◆You want to return an immutable collection
 - ◆You want a threadsafe collection
- Wrapper implementations add some functionality on top of what a collection offer
 - **◆**Unmodifiable
 - **♦**Synchronization
- Wrappers simply delegate all of their real work to a specified collection

Collection.unmodifiableSortedSet(SortedSet s)

77



CSD Univ. of Crete

Fall 2012

Unmodifiable wrappers

- The unmodifiable wrappers, rather than adding functionality to the wrapped collection, take functionality away
 - ◆ Any attempt to modify the collection generates an UnsupportedOperationException
- The unmodifiable wrappers have two main uses:
 - ◆ To make a collection immutable once it has been built
 - ◆ To allow "second-class citizens" read-only access to your data structures. You keep a reference to the backing collection, but hand out a reference to the wrapper

```
public static Collection unmodifiableCollection(Collection c);
public static Set unmodifiableSet(Set s);
public static List unmodifiableList(List list);
public static Map unmodifiableMap(Map m);
public static SortedSet unmodifiableSortedSet(SortedSet s);
public static SortedMap unmodifiableSortedMap(SortedMap m);
```



Thread Safety

- Collections, by default, are NOT thread-safe
 - ◆Design decision for performance and "conceptual weight"
- Solutions:
 - ◆Encapsulated Collections: In general, if the only access to a collection is through a thread-safe object, then that collection is safe
 - ◆Synchronized Collections: Wrapper implementations that synchronize all relevant methods
 - Factory methods inside the Collections class:

```
List list = Collections.synchronizedList(new
ArrayList(...));
```

- ◆Unmodifiable Collections: If an object can't be modified, it is threadsafe by definition
 - Factory methods inside the Collections class
 List list = Collections.unmodifiableList(new
 ArrayList(...));
- ◆Fail-fast iterators

79



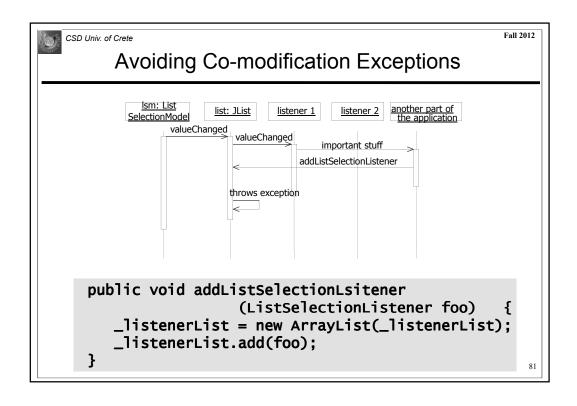
CSD Univ. of Crete

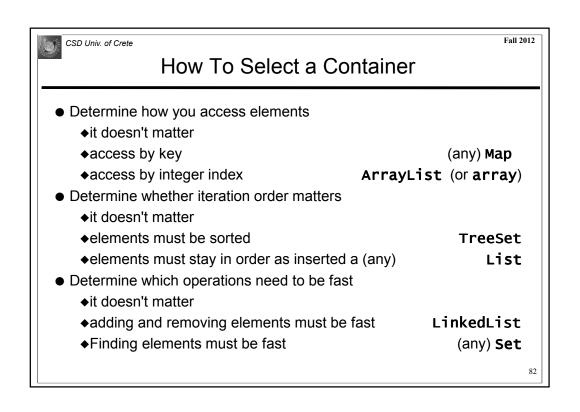
Fall 2012

"Fail-fast" Iterators

- What happens if the underlying collection changes during iteration?
 - ◆Iterators store enough information to detect concurrent modification
- Iterators throw exceptions if the underlying collection has changed
 - ◆If collection is modified during the life of an iterator, then that iterator fails immediately rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future
 - ◆Exception: the iterator's own add() and remove() methods work fine

```
final void checkForComodification() {
  if (modCount != expectedModCount)
     throw new ConcurrentModificationException();
}
```

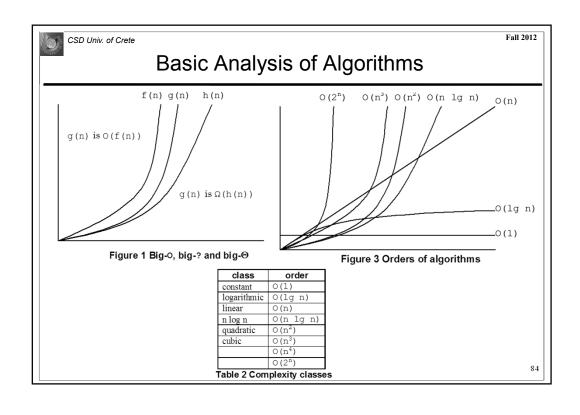






For Sets and Maps: Tree- or Hash-based?

- if hashCode is consistent with equals (all API classes) and iteration order doesn't matter
 - ◆use hash-based implementation
 - ◆otherwise use tree based implementation
- If both Hash and Tree based implementations are applicable, then the Hash based is usually the faster
- For tree based implementation:
 - ◆Is Comparable implemented or must Comparator be specified? (on objects for Sets, on keys for Maps)





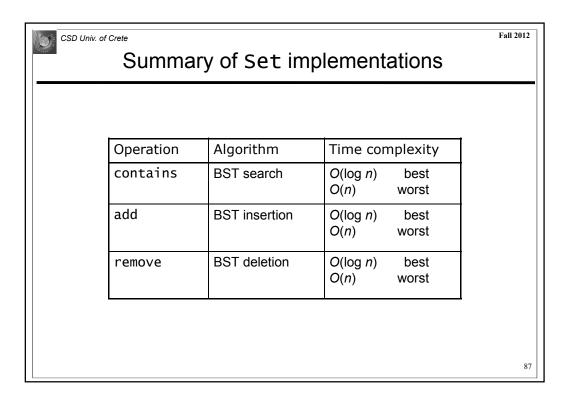
Operation	Member array representation	SLL representation	Boolean array representation
contains	O(log n)	O(n)	O(1)
add	O(n)	O(n)	O(1)
remove	O(n)	O(n)	O(1)
equals	O(n ₂)	O(n ₂)	O(m)
containsAll	O(n ₂)	O(n ₂)	O(m)
addAll	$O(n_1 + n_2)$	$O(n_1 + n_2)$	O(m)
removeAll	$O(n_1 + n_2)$	$O(n_1 + n_2)$	O(m)
retainAll	$O(n_1 + n_2)$	$O(n_1 + n_2)$	O(m)

85

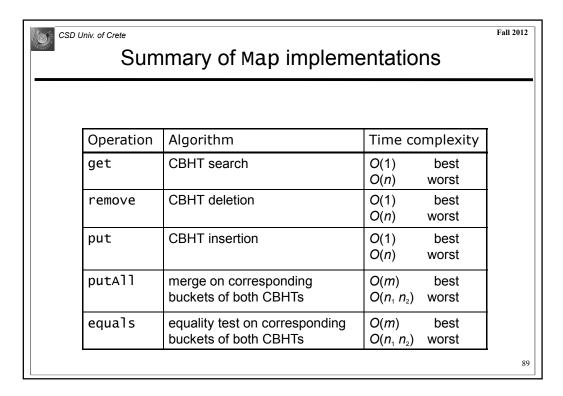
Summary of Set implementations

Operation	Algorithm	Time c	omplexity
contains	CBHT search	O(1) O(n)	best worst
add	CBHT insertion	O(1) O(n)	best worst
remove	CBHT deletion	O(1) O(n)	best worst

- Closed-bucket hash table (CBHT):
 - ◆ Each bucket may be occupied by several entries
 - ◆ Buckets are completely separate
- Simplest implementation: each bucket is an SLL



CSD Univ. of Cre	Summary	of Map in	nplement	ations	Fall 201
Operation	Key-indexed array repr- esentation	Array representation	SLL representation	BST representation	
get	O(1)	O(log n)	O(n)	O(log <i>n</i>) O(<i>n</i>)	best worst
remove	O(1)	O(n)	O(n)	O(log <i>n</i>) O(<i>n</i>)	best worst
put	O(1)	O(n)	O(n)	$O(\log n)$ O(n)	best worst
putAll	O(m)	$O(n_1+n_2)$	$O(n_1+n_2)$	$O(n_2 \log (n_1+n_2))$ best $O(n_1 n_2)$ worst	
equals	O(m)	O(n ₂)	O(n ₂)	$O(n_1 \log n_2) O(n_1 n_2)$	best worst



Fall 2012 CSD Univ. of Crete Summary of List implementations Operation SLL Array representation representation get 0(1) O(n)O(1) O(n)set add(int,Object) O(n) O(*n*) add(Object) O(1) 0(1) O(*n*) remove O(n)equals $O(n_2)$ $O(n_2)$ addA11 $O(n_2)$ $O(n_2)$