R·I·T

Computer Science II
4003-232-01 (Spring 2007-2008)

Week 5: Generics, Java Collection Framework

Richard Zanibbi
Rochester Institute of Technology

R·I·T

Generic Types in Java

(Ch. 21 in Liang)

What are ‘Generic Types’ or ‘Generics’?

Definition

- Reference type parameters for used to modify class, interface and method definitions (at compile time)
- Recall: “normal” (actual) method parameters: values from a primitive/reference type (at run-time for variables)
- Generic types define ‘macros:’ the class name replaces the type parameter in the source code (“search and replace”)

Syntax

<C> for parameter, use as C elsewhere (*C must be a class/interface*)

- public class Widget <C> { .... } // class definition
- Widget<String> widget = new Widget<String>(); // instantiation
- public <C> void test(C o1, int x) { C temp; .... } // method definition
- widget.<Integer>test(new Integer(5), 1); // invoking method

Purpose: Avoiding ‘Dangerous’ Polymorphism

Prevent run-time errors (*exceptions)* due to type errors (*casts)*

Example: Comparable Interface

Prior to JDK 1.5 (and Generic Types):

```
public interface Comparable {
    public int compareTo(Object o) }

Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

run-time error

JDK 1.5 (Generic Types):

```
public Interface Comparable<T> {
    public int compareTo(T o) }

Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

compile-time error

“Raw Types” and Associated Compiler Warnings

Raw Types (*for backward compatability)*

Classes with generic type parameters used without the type parameters defined

- e.g. Comparator c ~= Comparator<Object> c

Recommendation: always set generic type parameters for variable types

- e.g. Comparator<Date> c = new Date();

Compiler Warnings

– javac will give a warning about possibly unsafe operations (type errors) at run-time for raw types
  - use -Xlint:unchecked flag (or, -Xlint:all) for detailed messages.

– javac will not compile programs whose generic types cannot be properly defined
  - e.g. Max.java, Max1.java (pp. 699-700 in Liang)

R·I·T

Overview: Data Structures and Abstract Data Types

Storing Data in Java

Variables

Primitive type (int, double, boolean, etc.)

- Variable name refers to a memory location containing a primitive value

Reference type (Object, String, Integer, MyClass, etc.)

- Variable name refers to a memory location containing a reference value for data belonging to an object

Data Structure

A *formal* organization of a set of data (e.g. variables)

e.g. Arrays: variables of a given type in an integer-indexed sequence

- int intArray[ ] = {1, 2}; int a = intArray[0]; intArray[1] = 5;

e.g. Objects: data member names used to index variables

- player.name, player.hits, player.team ... player.hits = 100;

Abstract Data Types (ADTs)

Purpose

Define interfaces for complex data structures

- Hide *(abstract)* implementation details of operations that query and update
- Define operations to be *independent of the element type* (Java: “generic”)

Some Common ADTs

List: Sequence of elements. Elements may be inserted or removed from any position in the list

Stack: List with last-in, first-out (LIFO) behaviour (“most recent”)
e.g. call stack

Queue: List with first-in, first-out (FIFO) (“in-order”)
e.g. lining up at a fast-food restaurant or bank

Common ADTs, Cont'd

Tree: Graph with directed edges, each node has one parent (except root), no cycles.

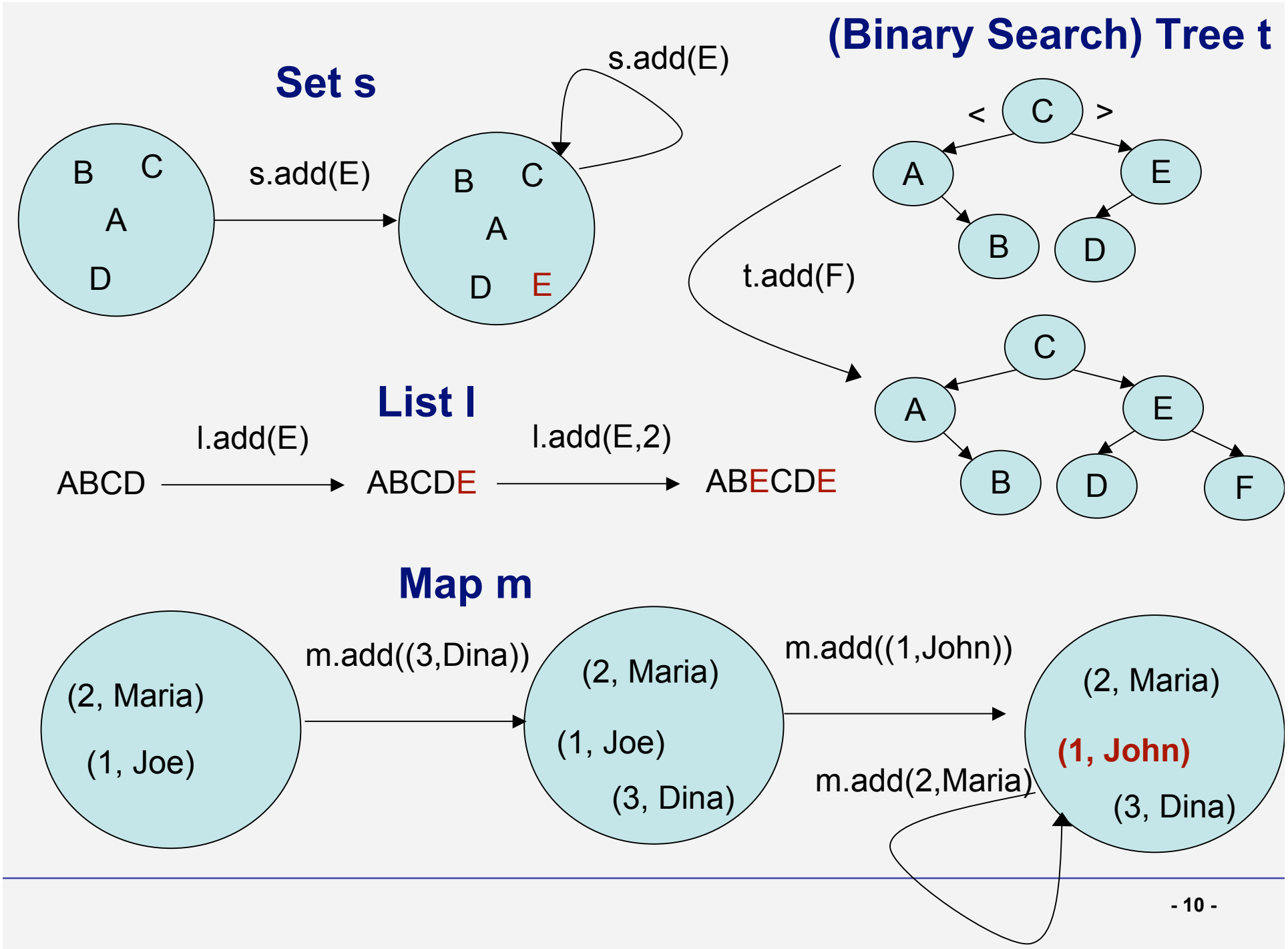
- A decision tree representing possible moves in a game of tic-tac-toe.

Set: Unordered group of unique items

- Students in a class, the set of words in a text file

Map: Set of entries, each with unique key and (possibly non-unique) value

- Student grade sheet: (StudentId, Grade)
- Frequency of words in a text file (Word, Count)


Set s
s.add(E)
B C A D
s.add(E)
B C A D E
(Binary Search) Tree t
< C >
A E
B D
t.add(F)
C
A E
B D F
List l
l.add(E)
l.add(E,2)
ABCD
ABCDE
ABECDE
Map m
(2, Maria)
(1, Joe)
m.add((3,Dina))
(2, Maria)
(1, Joe)
(3, Dina)
m.add((1,John))
(2, Maria)
(1, John)
(3, Dina)
m.add(2,Maria)

Example: Implementing Abstract Data Types

List ADT

Represents series of elements, insertion and deletion of elements

Some Possible Implementations:

- Arrays:
  L.add(E) would copy E at the end of the array, L.get(4) returns 5th item in array
- Linked List: (objects forming a chain of references)
  L.add(E) would create a link from last node to a new node for E ; l.get(4) traverses the graph and then returns the 5th item

Choosing an Implementation for an ADT

Depending on common operations, some better than others

- Retrieving elements in list faster for array implementation
- Inserting, deleting elements faster for linked list implementation in general case

Ordering in ‘Unordered’ Sets and Maps

‘Unordered’ in Theory vs. in Code

In practice, an ordering of elements is used to implement a set, as memory and files store data as lists of bytes.

Sets

By definition, a set is an unordered group of unique elements

Maps

By definition, a map is a set of (key,value) pairs, where all keys are unique.

Ordering Sets and Maps

We can order the storage of set elements by:

1. The order in which elements are added (e.g. in a list)
2. The values of keys or data elements themselves (e.g. using a binary search tree)
3. A value computed for each element (“hash code”) that determines where an element is stored (e.g. in a “hash table”, a sophisticated ADT built on arrays); for maps, hash code computed using key value

Exercise: Generics and ADTs

Part A

1. In one sentence, what is a generic type?
2. What errors are generic types designed to prevent?
3. Which javac flag will show details for (type) unsafe operations?
4. What do the following represent:
   a) <? extends MyClass>
   b) <? super YourClass>
   c) <E extends Comparator<E>>
5. Write a java class *GenX* which has a generic type parameter *T*, a public data member *identity* of type *T*, and a constructor that takes an initial value for *identity.* Add a main method that constructs one *GenX* object using type *String*, and another using type *Integer*.

Part B

1. What is an abstract data type?
2. How is a list different from a set?
3. How are elements stored in a binary search tree (BST)?
4. In what ways can we order the elements of a set, or pairs of a map?
5. Are sets and map elements/pairs ordered in their ADT definitions?

R·I·T

ADTs in Java: The Java Collections Framework

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.)

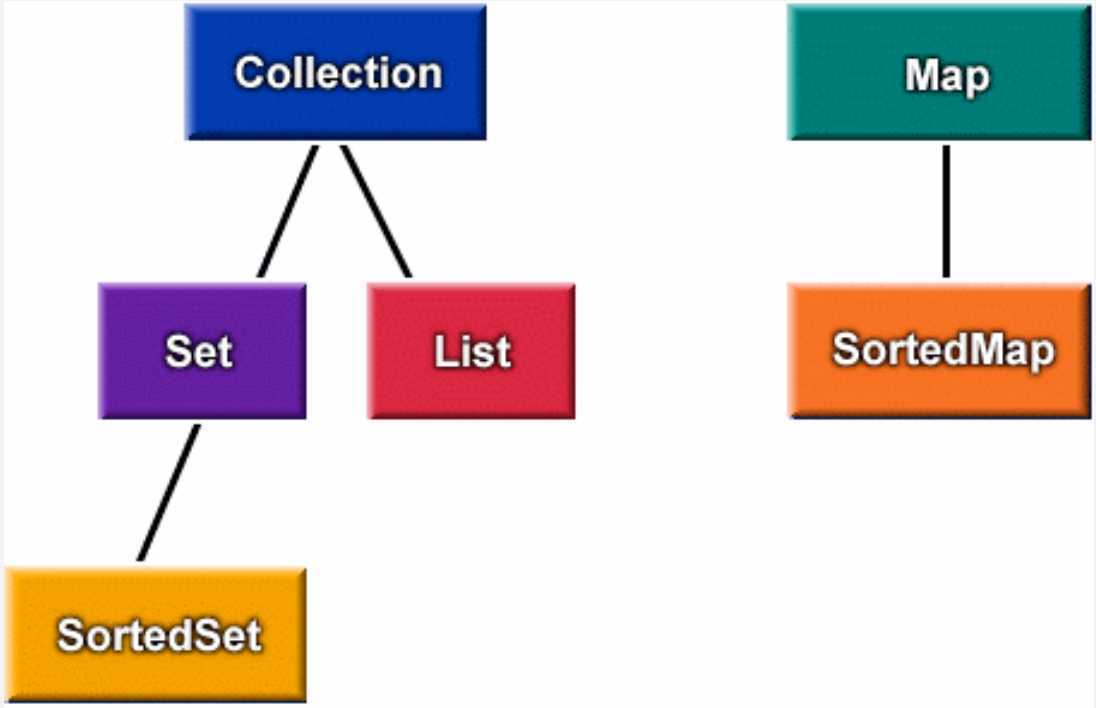
Maps

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

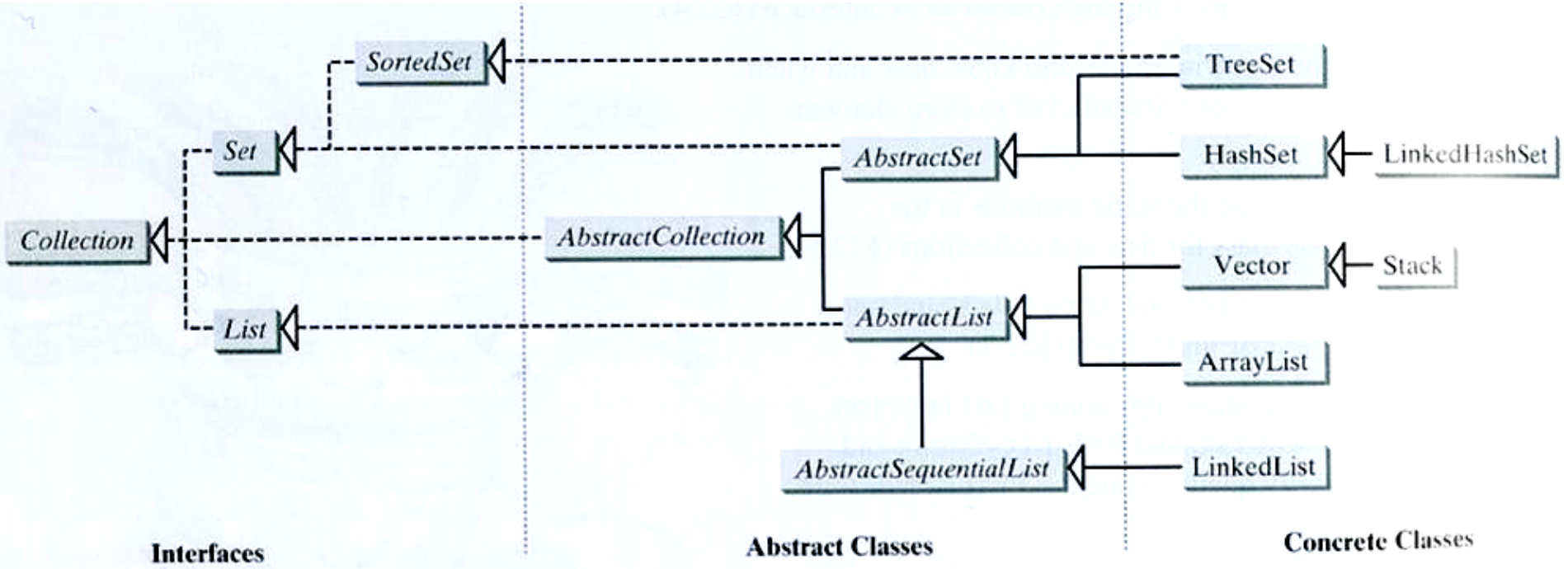


FIGURE 22.1 `Set` and `List` are subinterfaces of `Collection`.

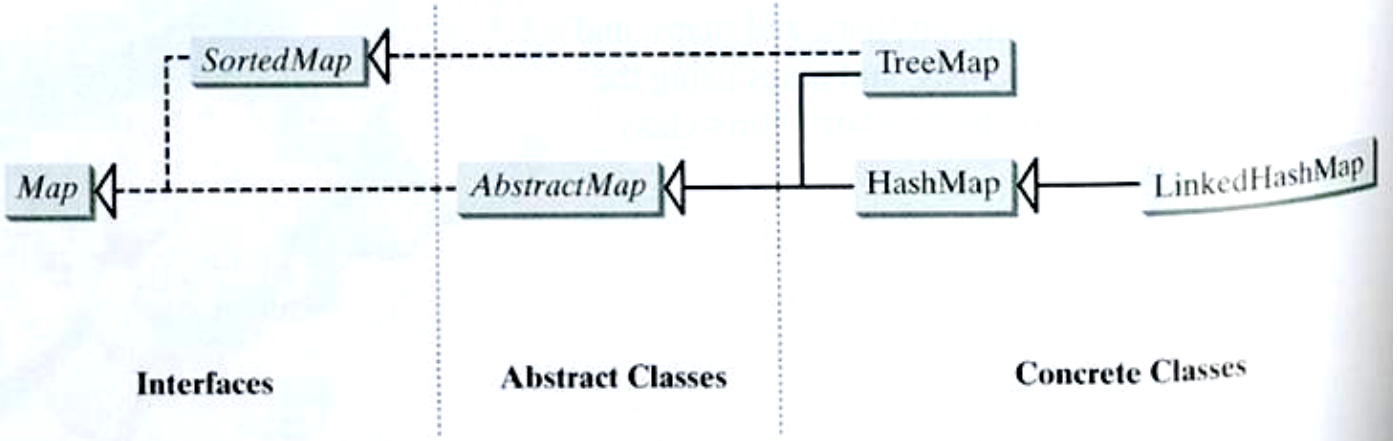


FIGURE 22.2 An instance of `Map` stores a group of objects and their associated keys.

Implementation Classes

(*slide derived from: Carl Reynolds*)

Interface	Implementation				Historical
`Set`	`HashSet`		`TreeSet`	`LinkedHashSet`	
`List`		`ArrayList`		`LinkedList`	`Vector` `Stack`
`Map`	`HashMap`		`TreeMap`	`LinkedHashMap`	`HashTable` `Properties`

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

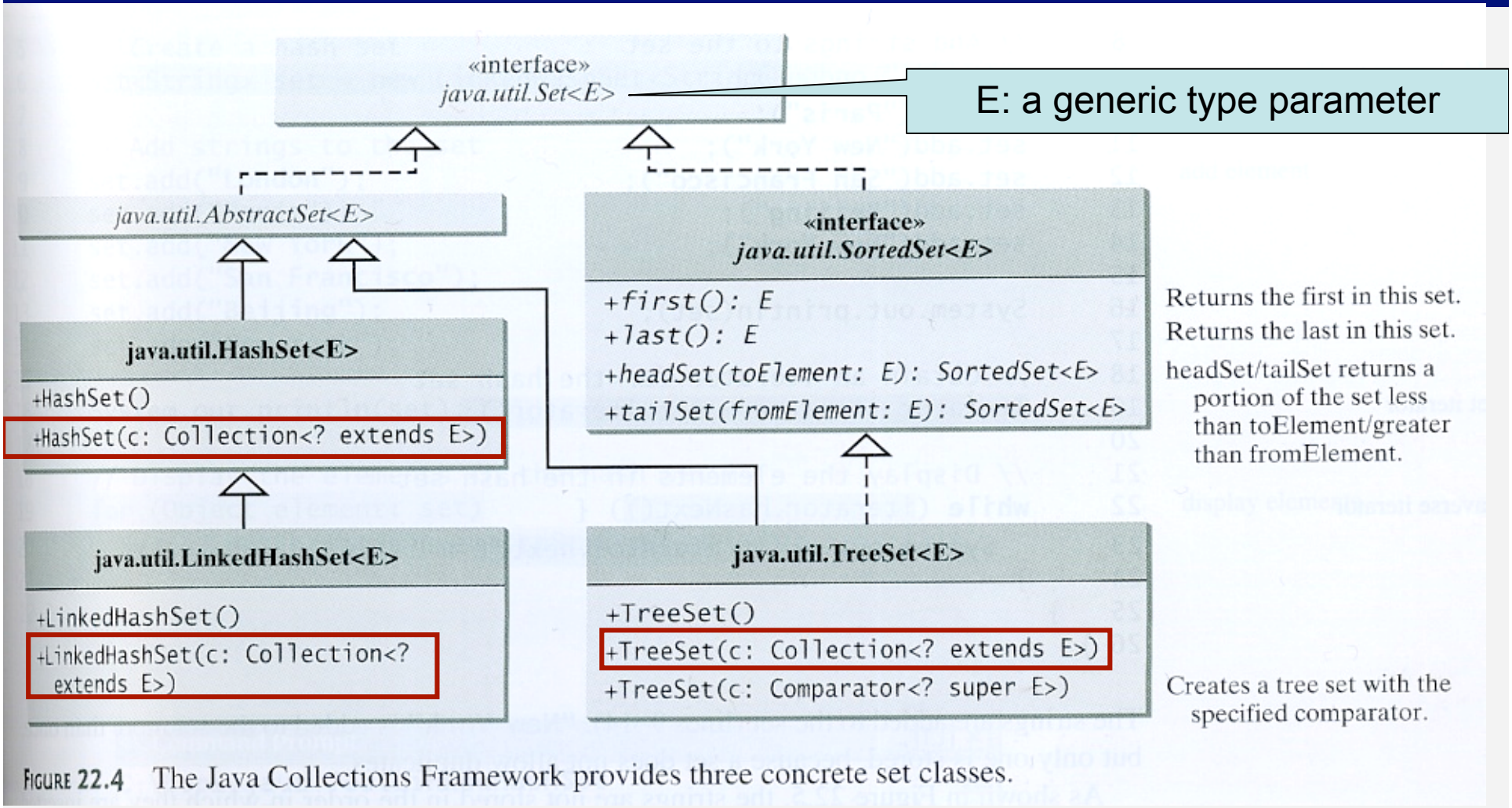


FIGURE 22.4 The Java Collections Framework provides three concrete set classes.

E: a generic type parameter

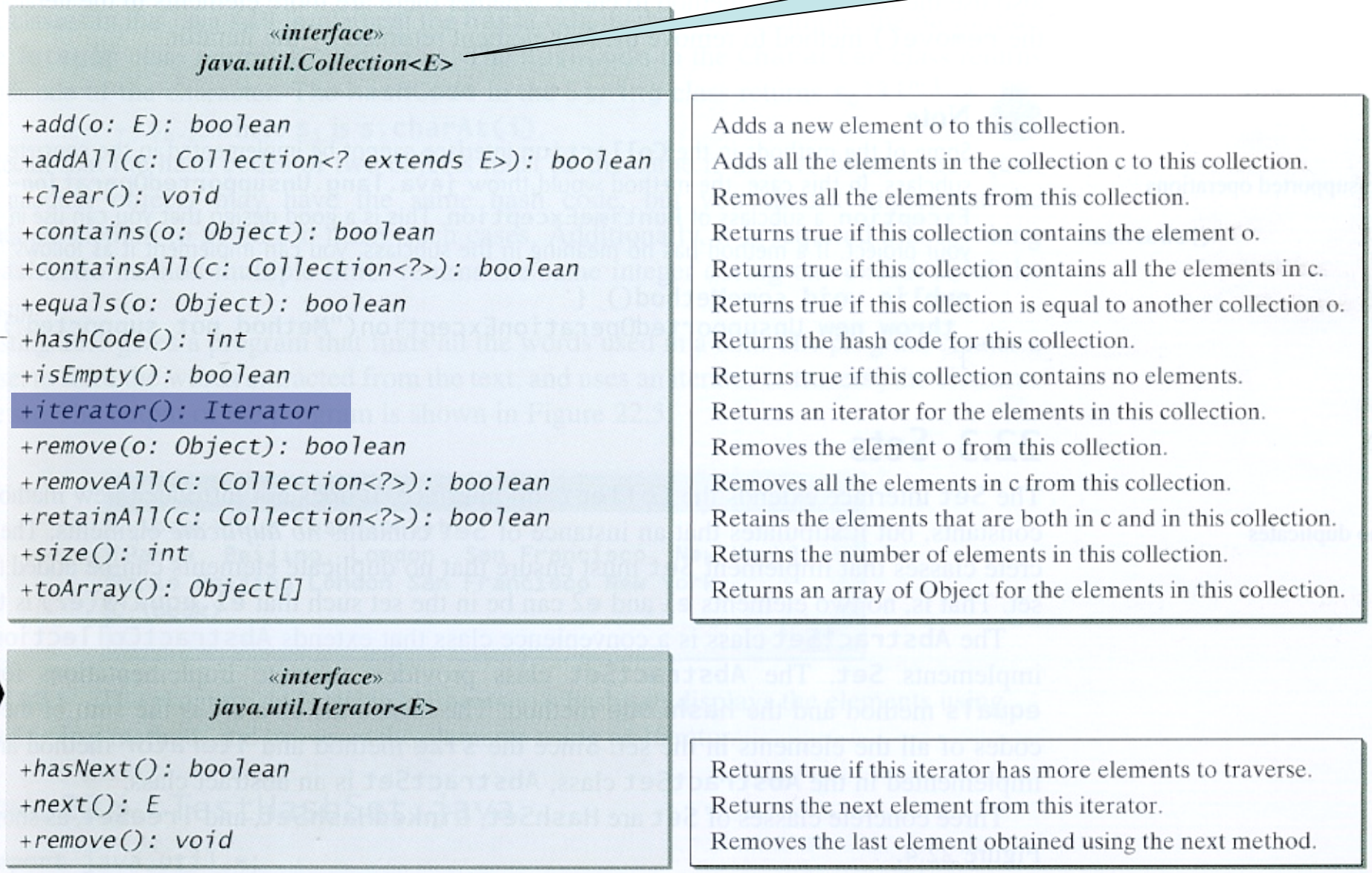


FIGURE 22.3 The `Collection` interface contains the methods for manipulating the elements in a collection, and each collection object contains an iterator for traversing elements in the collection.

HashSet
(Example: TestHashSet.java, p. 717)

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, p. 718)

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
- 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

```
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() );
  }
}
```