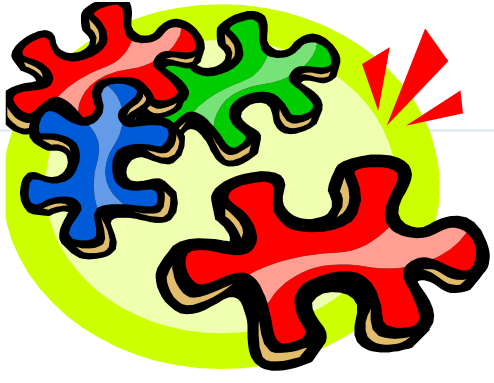
JAVA COLLECTION FRAMEWORK & SETS

Ch07.4-5 & Ch10.5



Presentation for use with the textbook

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2. Data Structures Abstraction and Design Using Java, 2nd Edition by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

© 2014 Goodrich, Tamassia, Goldwasser

LIST ADT REVIEWED

- We learned about part of the **List** ADT
- The classes that implement the `List` interface are all *indexed* collections
  - An index or subscript is associated with each element
  - The element's index often reflects the relative order of its insertion into the list
  - Searching for a particular value in a list is generally O(*n*)
  - An exception is a binary search of a sorted object, which is O(log *n*)

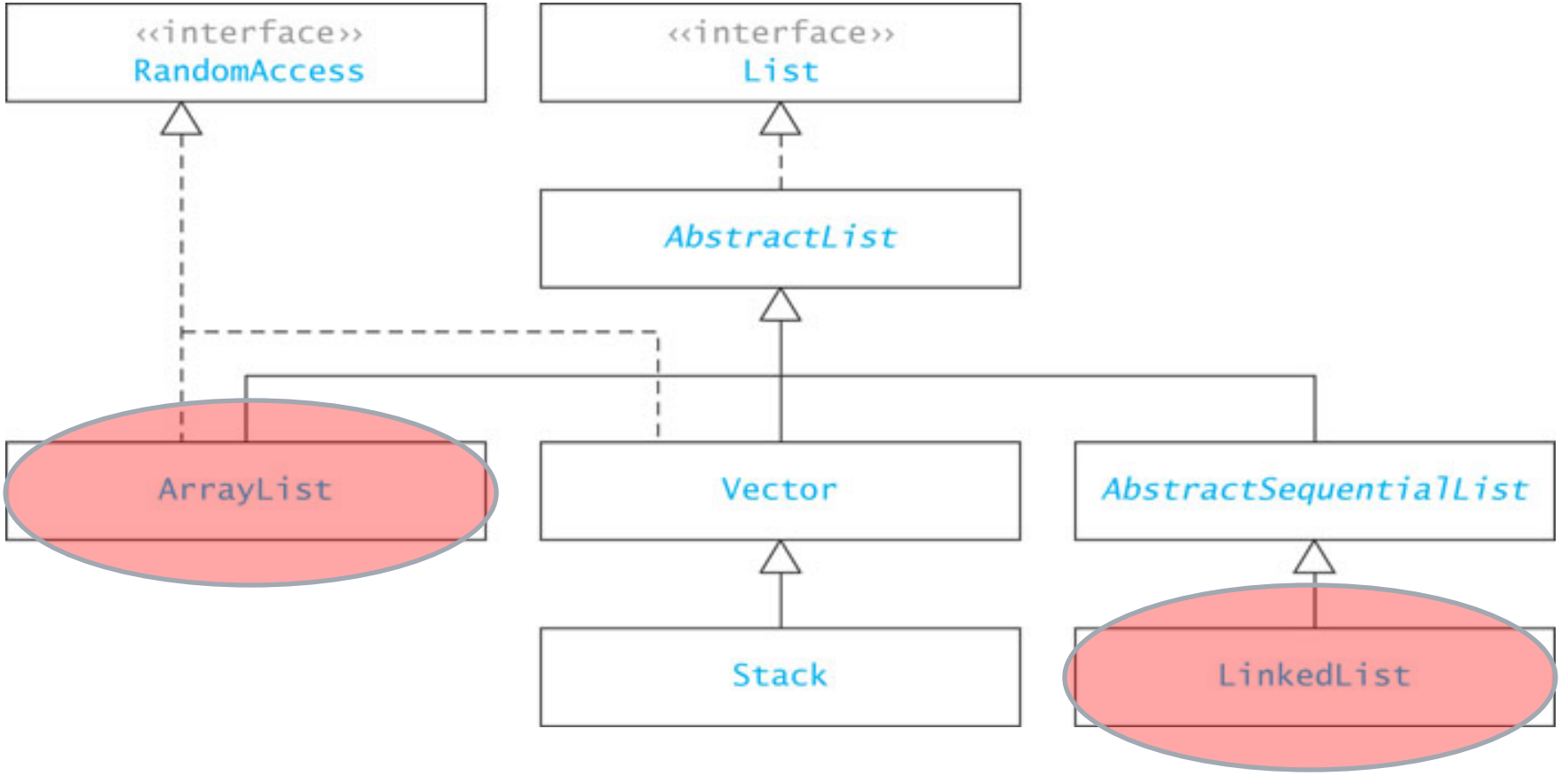
LIST ADT REVIEWED

```
1  /** A simplified version of the java.util.List interface. */
2  public interface List<E> {
3    /** Returns the number of elements in this list. */
4    int size( );
5
6    /** Returns whether the list is empty. */
7    boolean isEmpty( );
8
9    /** Returns (but does not remove) the element at index i. */
10   E get(int i) throws IndexOutOfBoundsException;
11
12   /** Replaces the element at index i with e, and returns the replaced element. */
13   E set(int i, E e) throws IndexOutOfBoundsException;
14
15   /** Inserts element e to be at index i, shifting all subsequent elements later. */
16   void add(int i, E e) throws IndexOutOfBoundsException;
17
18   /** Removes/returns the element at index i, shifting subsequent elements earlier. */
19   E remove(int i) throws IndexOutOfBoundsException;
20 }
```

Code Fragment 7.1: A simple version of the List interface.

© 2014 Goodrich, Tamassia, Goldwasser

JAVA.UTIL.LIST INTERFACE AND ITS IMPLEMENTERS



Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

METHODS OF THE ARRAYLIST CLASS

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `ArrayList`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `ArrayList`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `ArrayList`.
`public E remove(int index)`	Returns and removes the item at position `index` and shifts the items that follow it to fill the vacated space.

public boolean isEmpty()

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

THE LINKEDLIST CLASS

Method	Behavior
`public void add(int index, E obj)`	Inserts object `obj` into the list at position `index`.
`public void addFirst(E obj)`	Inserts object `obj` as the first element of the list.
`public void addLast(E obj)`	Adds object `obj` to the end of the list.
`public E get(int index)`	Returns the item at position `index`.
`public E getFirst()`	Gets the first element in the list. Throws `NoSuchElementException` if the list is empty.
`public E getLast()`	Gets the last element in the list. Throws `NoSuchElementException` if the list is empty.
`public boolean remove(E obj)`	Removes the first occurrence of object `obj` from the list. Returns `true` if the list contained object `obj`; otherwise, returns `false`.
`public int size()`	Returns the number of objects contained in the list.

public boolean isEmpty()

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

ITERATORS

THE ITERATOR

- An *iterator* is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.
- An Iterator object for a list starts at the first node
- The programmer can move the Iterator by calling its next method
- The Iterator stays on its current list item until it is needed
- An Iterator traverses in O(n) while a list traversal using get() calls in a linked list is O(n^2)

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

ITERATOR INTERFACE

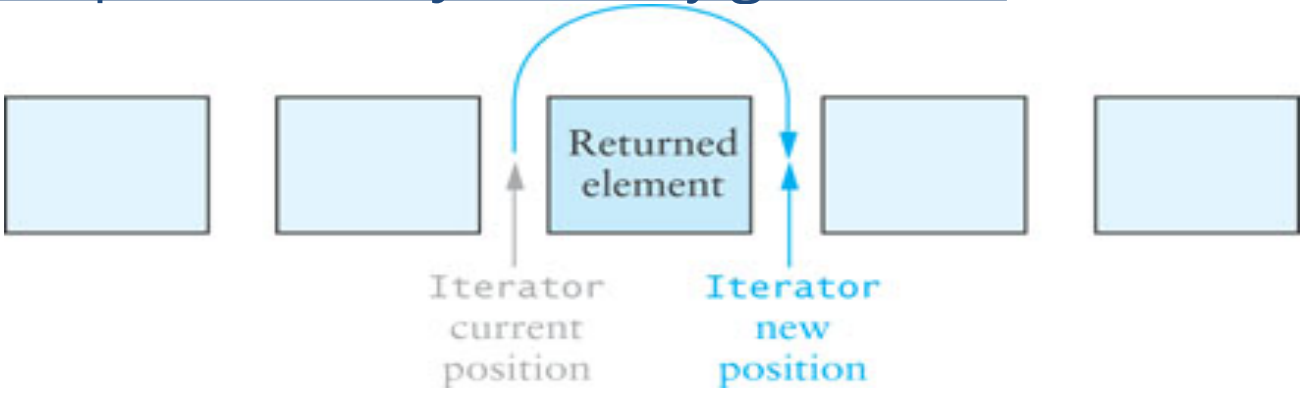
- The **`List`** interface declares the method **`iterator`** which returns an **`Iterator`** object that iterates over the elements of that list
- Java defines the *java.util.Iterator* interface with the following two methods:

hasNext( ):	Returns true if there is at least one additional element in the sequence, and false otherwise.
next( ):	Returns the next element in the sequence.
remove( ):	Removes from the collection the element returned by the most recent call to next( ). Throws an IllegalStateException if next has not yet been called, or if remove was already called since the most recent call to next.

© 2014 Goodrich, Tamassia, Goldwasser

ITERATOR INTERFACE (CONT.)

- An **`Iterator`** is conceptually *between* elements; it does not refer to a particular object at any given time



- A single iterator instance supports only one pass through a collection
  - there is no way to "reset" the iterator back to the beginning of the sequence.

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

EXAMPLE OF IMPLEMENTING ITERATOR

- Nested ArrayIterator Class inside ArrayList

```
1   // ---------- nested ArrayIterator class ----------
2     /**
3      * A (nonstatic) inner class. Note well that each instance contains an implicit
4      * reference to the containing list, allowing it to access the list's members.
5      */
6     private class ArrayIterator implements Iterator<E> {
7       private int j = 0;                          // index of the next element to report

14      public boolean hasNext( ) {

22      public E next( ) throws NoSuchElementException
33      public void remove( ) throws IllegalStateException
```

- Additional Method of ArrayList

```
41    /** Returns an iterator of the elements stored in the list. */
42    public Iterator<E> iterator( ) {
43      return new ArrayIterator( );        // create a new instance of the inner class
44    }
```

© 2014 Goodrich, Tamassia, Goldwasser

ITERABLE INTERFACE,

- To provide greater standardization, Java defines another parameterized interface, named **Iterable**, that includes the following single method:

  iterator( ): Returns an iterator of the elements in the collection.

- Each call to **iterator( )** returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.
- An instance of a typical collection class in Java, such as an ArrayList, is *iterable* (but not itself an *iterator*);

```
41    /** Returns an iterator of the elements stored in the list. */
42    public Iterator<E> iterator( ) {
43      return new ArrayIterator( );        // create a new instance of the inner class
44    }
```

© 2(

ITERATOR INTERFACE (CONT.)

- While Loop with Iterator

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
  ElementType variable = iter.next();
  loopBody                                    // may refer to "variable"
}
```

- EX> process all items in `List<Integer>` through an `Iterator`

```
Iterator<Integer> iter =
aList.iterator();
while (iter.hasNext()) {
  int value = iter.next();
  // Do something with value
}
```

ITERATORS AND REMOVING ELEMENTS

- You can use the `Iterator remove()` method to remove items from a list as you access them
- `remove()` deletes the most recent element returned
- You must call `next()` before each `remove();` otherwise, an `IllegalStateException` will be thrown
- `LinkedList.remove` vs. `Iterator.remove`:
  - `LinkedList.remove` must walk down the list each time, then remove, so in general it is $O(n^2)$
  - `Iterator.remove` removes items without starting over at the beginning, so in general it is O(n)

ITERATORS AND REMOVING ELEMENTS (CONT.)

- To remove all elements from a list of type `Integer` that are divisible by a particular value:

```
public static void removeDivisibleBy(LinkedList<Integer>
                                                aList, int div) {
  Iterator<Integer> iter = aList.iterator();
  while (iter.hasNext()) {
    int nextInt = iter.next();
    if (nextInt % div == 0) {
      iter.remove();
    }
  }
}
```

ENHANCED FOR STATEMENT

- Java 5.0 introduced an enhanced `for` statement
- The enhanced `for` statement creates an `Iterator` object and implicitly calls its `hasNext` and `next` methods
- Other `Iterator` methods, such as `remove`, are not available

ENHANCED FOR STATEMENT (CONT.)

- The while statement in the “for-each” loop syntax

```
for (ElementType variable : collection) {
  loopBody                                  // may refer to "variable"
}
```

- The following code counts the number of times `target` occurs in `myList` (type `LinkedList<String>`)

```
count = 0;
for (String nextStr : myList) {
  if (target.equals(nextStr)) {
    count++;
  }
}
```

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

ENHANCED FOR STATEMENT (CONT.)

- In list `myList` of type `LinkedList<Integer>`, each `Integer` object is automatically unboxed:

```
sum = 0;
for (int nextInt : myList) {
  sum += nextInt;
}
```

NOTE: the iterator's *remove* method cannot be invoked when using the for-each loop syntax.

ITERABLE INTERFACE REVISITED

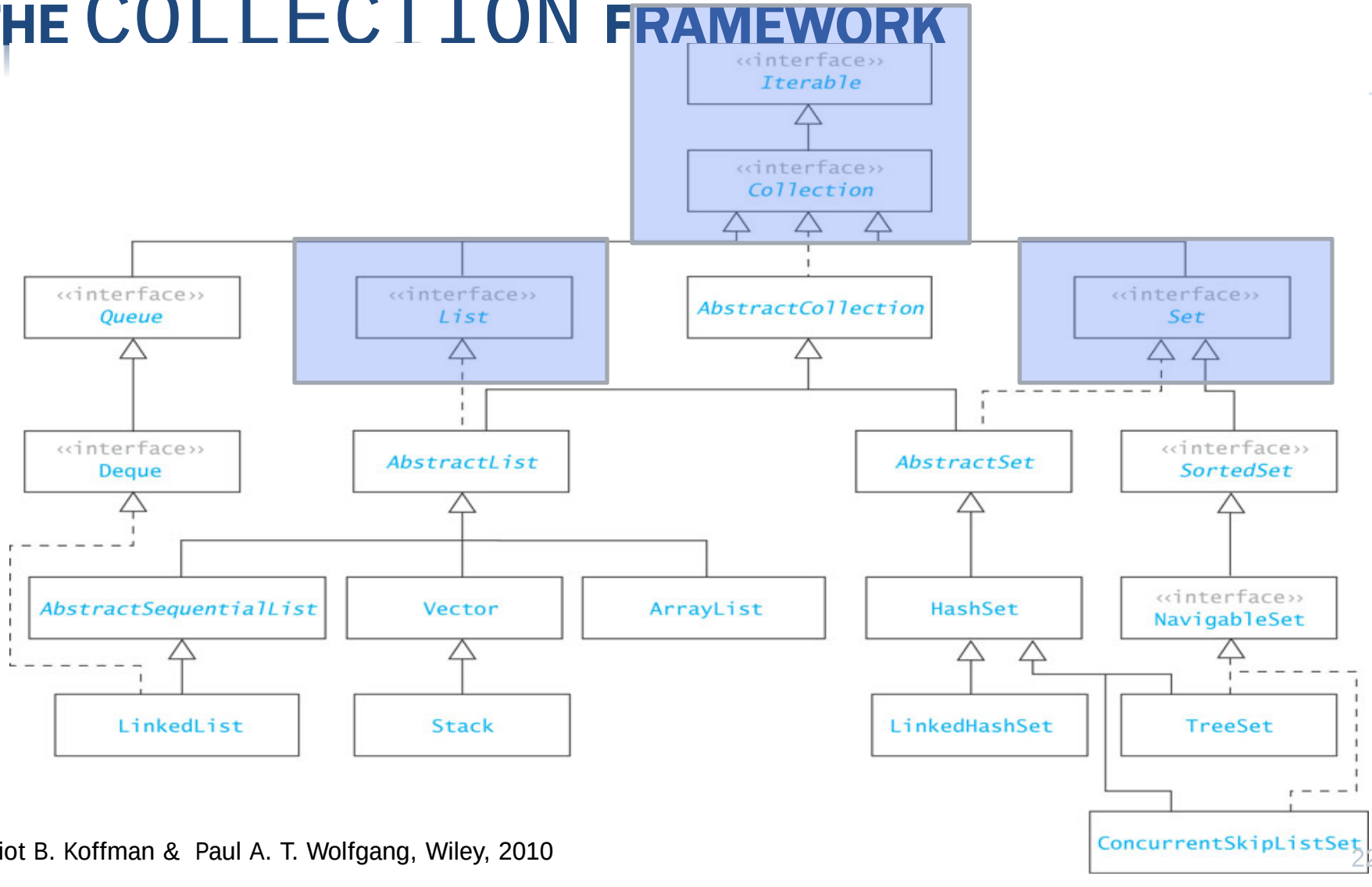
- Each class that implements the `List` interface must provide an `iterator` method
- The `Collection` interface extends the `Iterable` interface
- All classes that implement the `List` interface (a subinterface of `Collection`) must provide an `iterator` method
- Allows use of the Java 5.0 *for-each* loop

```
public interface Iterable<E> {
    /** returns an iterator over the elements
      * in this collection. */
  Iterator<E> iterator();
}
```

Section 2.9

THE COLLECTIONS FRAMEWORK DESIGN

THE COLLECTION FRAMEWORK



Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

THE COLLECTION INTERFACE

- Specifies a subset of methods in the `List` interface, specifically excluding
  - `add(int, E)`
  - `get(int)`
  - `remove(int)`
  - `set(int, E)`

  but including
  - `add(E)`
  - `remove(Object)`
  - the `iterator` method `iterator()`

COMMON FEATURES OF COLLECTIONS

- Collections
  - Grow as needed
  - Hold references to objects
  - Have at least two constructors:
    - one to create an empty collection and
    - one to make a copy of another collection

COMMON FEATURES OF COLLECTIONS (CONT.)

Method	Behavior
`boolean add(E obj)`	Ensures that the collection contains the object `obj`. Returns `true` if the collection was modified.
`boolean contains(E obj)`	Returns `true` if the collection contains the object `obj`.
`Iterator<E> iterator()`	Returns an `Iterator` to the collection.
`int size()`	Returns the size of the collection.

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

Section 2.9

SETS

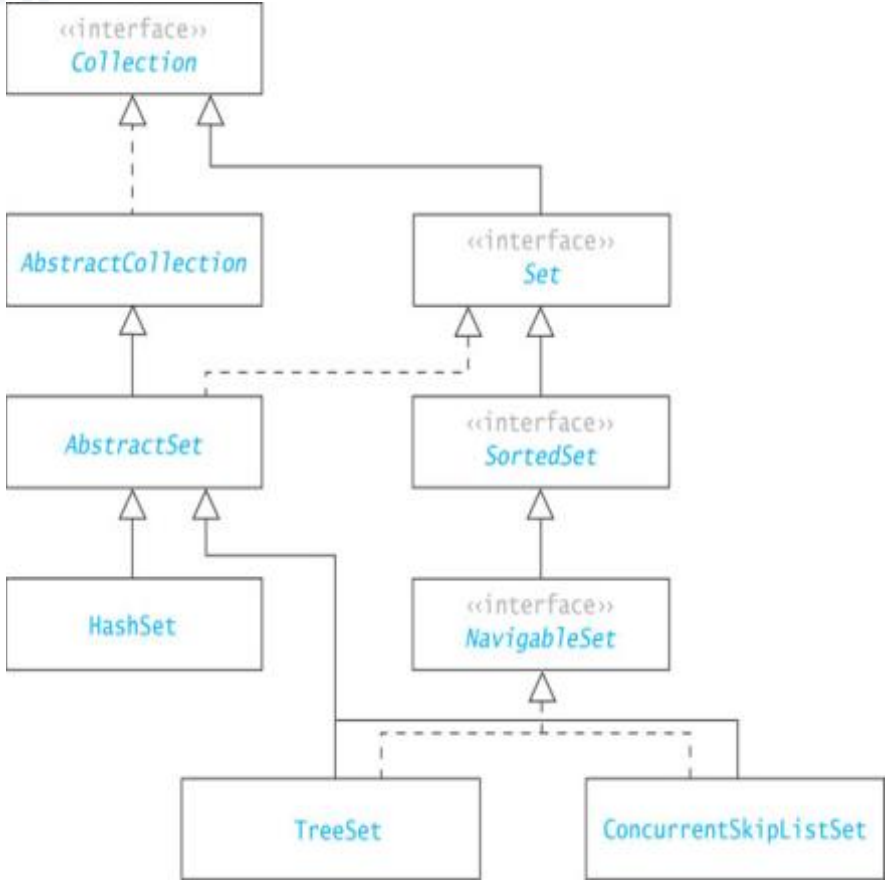
INTRODUCTION

- Consider another part of the Collection hierarchy: the Set interface
- A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.
  - Elements of a set are like keys of a map, but without any auxiliary values.
- A **multiset** (also known as a **bag**) is a set-like container that allows duplicates.

© 2014 Goodrich, Tamassia, Goldwasser

SETS AND THE SET INTERFACE

- **`Set`** objects
  - Are not indexed
  - Do not reveal the order of insertion of items
  - Enable efficient search and retrieval of information
  - Allow removal of elements without moving other elements around



Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

THE SET ABSTRACTION(CONT.)

- The **union** of two sets A, B is a set whose elements belong either to A or B or to both A and B.

  Example: {1, 3, 5, 7} ∪ {2, 3, 4, 5} is {1, 2, 3, 4, 5, 7}
- The **intersection** of sets A, B is the set whose elements belong to both A and B.

  Example: {1, 3, 5, 7} ∩ {2, 3, 4, 5} is {3, 5}
- The **difference** of sets A, B is the set whose elements belong to A but not to B.

  Examples: {1, 3, 5, 7} – {2, 3, 4, 5} is {1, 7}; {2, 3, 4, 5} – {1, 3, 5, 7} is {2, 4}
- Set A is a **subset** of set B if every element of set A is also an element of set B.

  Example: {1, 3, 5, 7} ⊂ {1, 2, 3, 4, 5, 7} is true

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

THE SET INTERFACE AND METHODS

- Required methods:
  - testing set membership,
  - testing for an empty set,
  - determining set size, and
  - creating an iterator over the set
- Optional methods:
  - adding an element (not allow duplicate items) and
  - removing an element
- Constructors to enforce the “no duplicate members” criterion
  - The add method does not allow duplicate items to be inserted

Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010

SET ADT

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e:\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e:\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

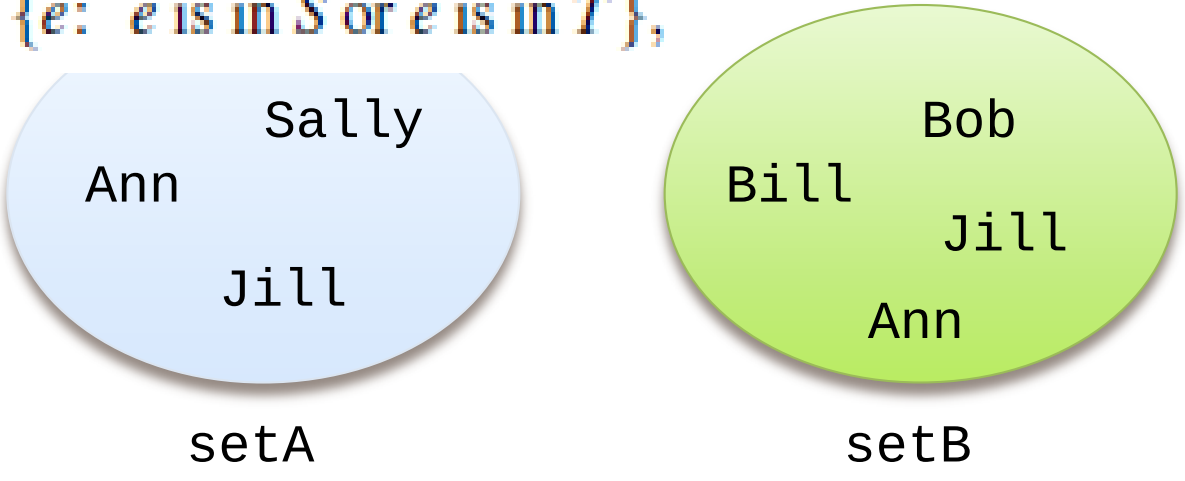
retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

© 2014 Goodrich, Tamassia, Goldwasser

THE SET INTERFACE AND METHODS

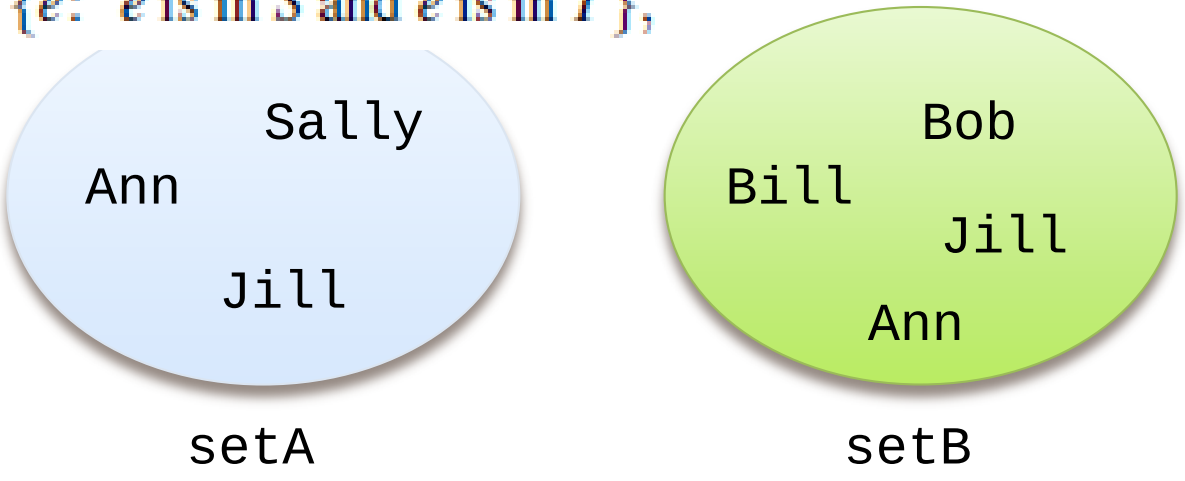
$$S \cup T = \{e: \ e \text{ is in } S \text{ or } e \text{ is in } T\},$$




```
setA.addAll(setB);
System.out.println(setA);
Outputs:
[Bill, Jill, Ann, Sally, Bob]
```

THE SET INTERFACE AND METHODS(CONT.)

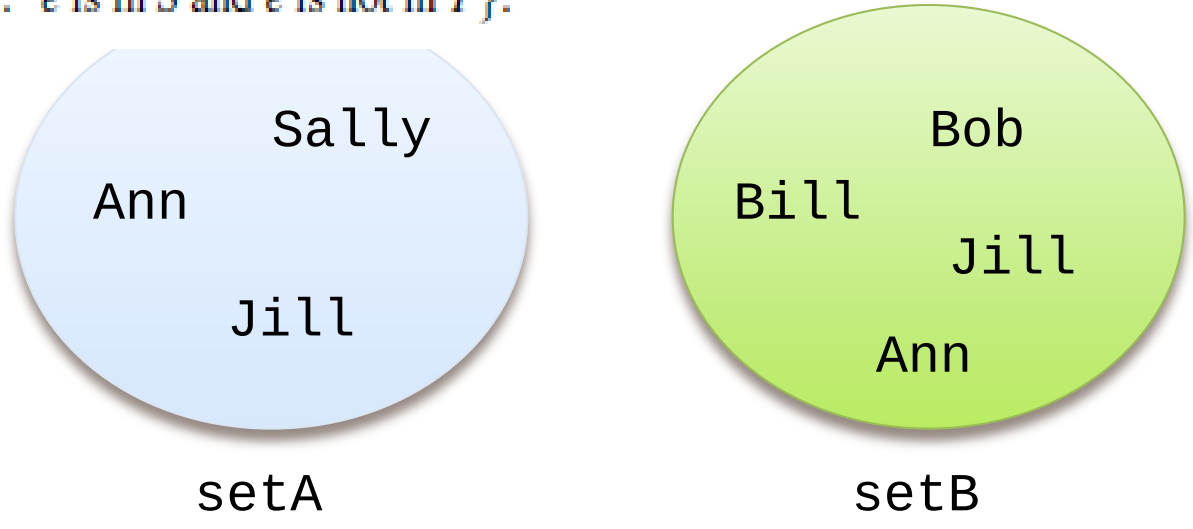
$$S \cap T = \{e: e \text{ is in } S \text{ and } e \text{ is in } T\},$$




```
setACopy.retainAll(setB);
System.out.println(setACopy);
Outputs:
[Jill, Ann]
```

THE SET INTERFACE AND METHODS(CONT.)

$S - T = \{e: e \text{ is in } S \text{ and } e \text{ is not in } T\}.$




```
setACopy.removeAll(setB);
System.out.println(setACopy);
Outputs:
[Sally]
```

COMPARISON OF LISTS AND SETS

- Collections implementing the `Set` interface may contain only unique elements
- Unlike the `List.add` method, the `Set.add` method returns `false` if you attempt to insert a duplicate item
- Unlike a `List`, a `Set` does not have a `get` method—elements cannot be accessed by index

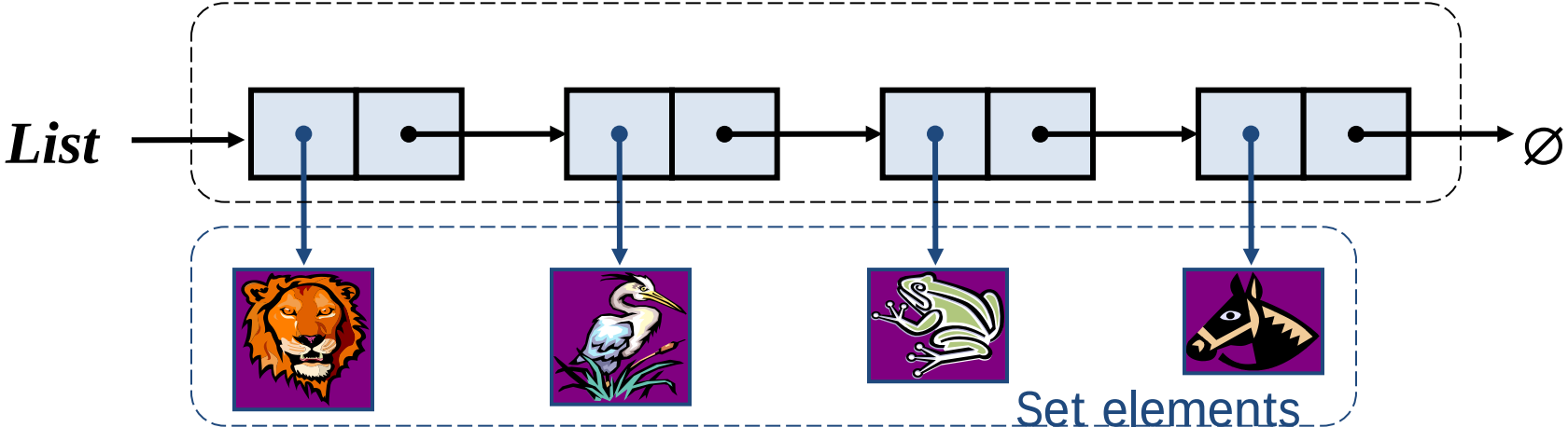
COMPARISON OF LISTS AND SETS (CONT.)

- You can iterate through all elements in a `Set` using an `Iterator` object, but the elements will be accessed in arbitrary order

```
for (String nextItem : setA) {
  //Do something with nextItem
  …
}
```

STORING A SET IN A LIST

- We can implement a set with a list
- The space used is $O(n)$



© 2014 Goodrich, Tamassia, Goldwasser