CMSC 202H

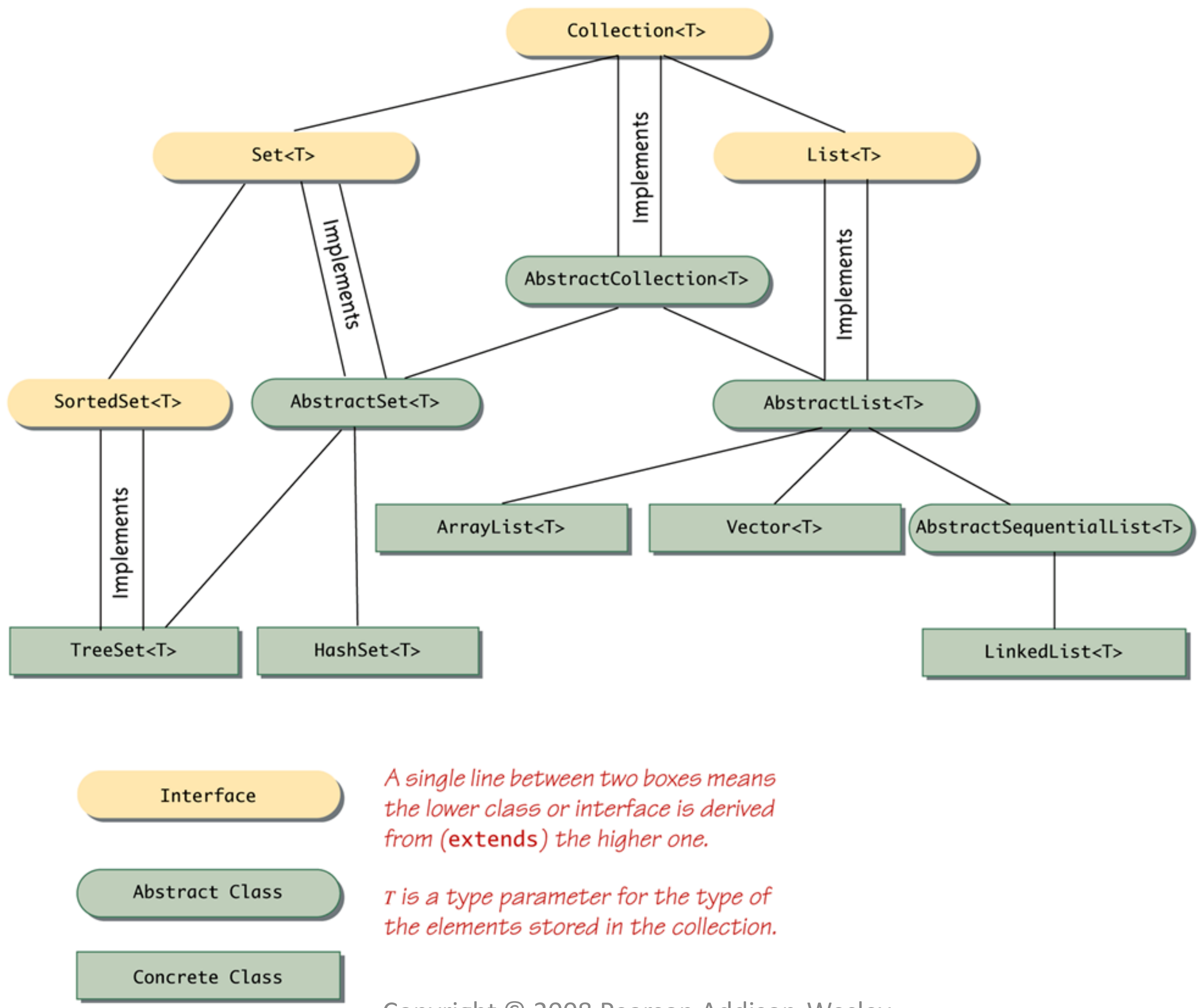
Containers and Iterators

Container Definition

- A “container” is a data structure whose purpose is to hold objects.
- Most languages support several ways to hold objects
  - Arrays are compiler-supported containers
  - Language libraries provide a set of classes
- Java supports two primary kinds of container interfaces
  - **Collections** contain a sequence of individual elements
  - **Maps** contain a group of key-value object pairs

The `Collection` Landscape

Display 16.1 The Collection Landscape



A single line between two boxes means the lower class or interface is derived from (`extends`) the higher one.

T is a type parameter for the type of the elements stored in the collection.

Wildcards

- Classes and interfaces in the collection framework can have parameter type specifications that do not fully specify the type plugged in for the type parameter
  - Because they specify a wide range of argument types, they are known as *wildcards*

```
public void method(String arg1, ArrayList<?> arg2)
```

  - In the above example, the first argument is of type `String`, while the second argument can be an `ArrayList<T>` with any base type

Wildcards

- A bound can be placed on a wildcard specifying that the type used must be an ancestor type or descendent type of some class or interface
  - The notation **`<? extends String>`** specifies that the argument plugged in be an object of any descendent class of **`String`**
  - The notation **`<? super String>`** specifies that the argument plugged in be an object of any ancestor class of **`String`**

Collection<T> Interface

- The Collection<T> interface generalizes the concept of a *sequence* of elements.
- Basic Collection<T> operations include
  - No-argument and copy constructors
  - boolean contains(Object x) – returns true if at least one instance of x is in the collection
  - boolean containsAll(Collection<?> targets) – returns true if all targets are contained in the calling collection object
  - boolean equals(Object x) – This is equals for the collection, not the elements in the collection. Intuitive meaning.
  - Object[ ] toArray( ) – returns an array containing all of the elements
  - boolean add(T element) – ensures that the calling collection object contains the specified element. (optional)
  - boolean addAll(Collection<? extends T> collectionToAdd) – ensures that the calling collection object contains all elements of collectionToAdd (optional)
  - boolean remove(T element) – removes a single instance of the element from the calling collection object (optional)
  - boolean removeAll(Collection<?> collectionToRemove) – removes all elements contained in collectionToRemove from the calling collection object (optional)
  - void clear( ) – removes all elements from the calling collection object (optional)

Collection Relationships

- There are a number of different predefined classes that implement the `Collection<T>` interface
  - Programmer defined classes can implement it also
- A method written to manipulate a parameter of type `Collection<T>` will work for all of these classes, either singly or intermixed
- There are two main interfaces that extend the `Collection<T>` interface
  - The `Set<T>` interface and
  - The `List<T>` interface

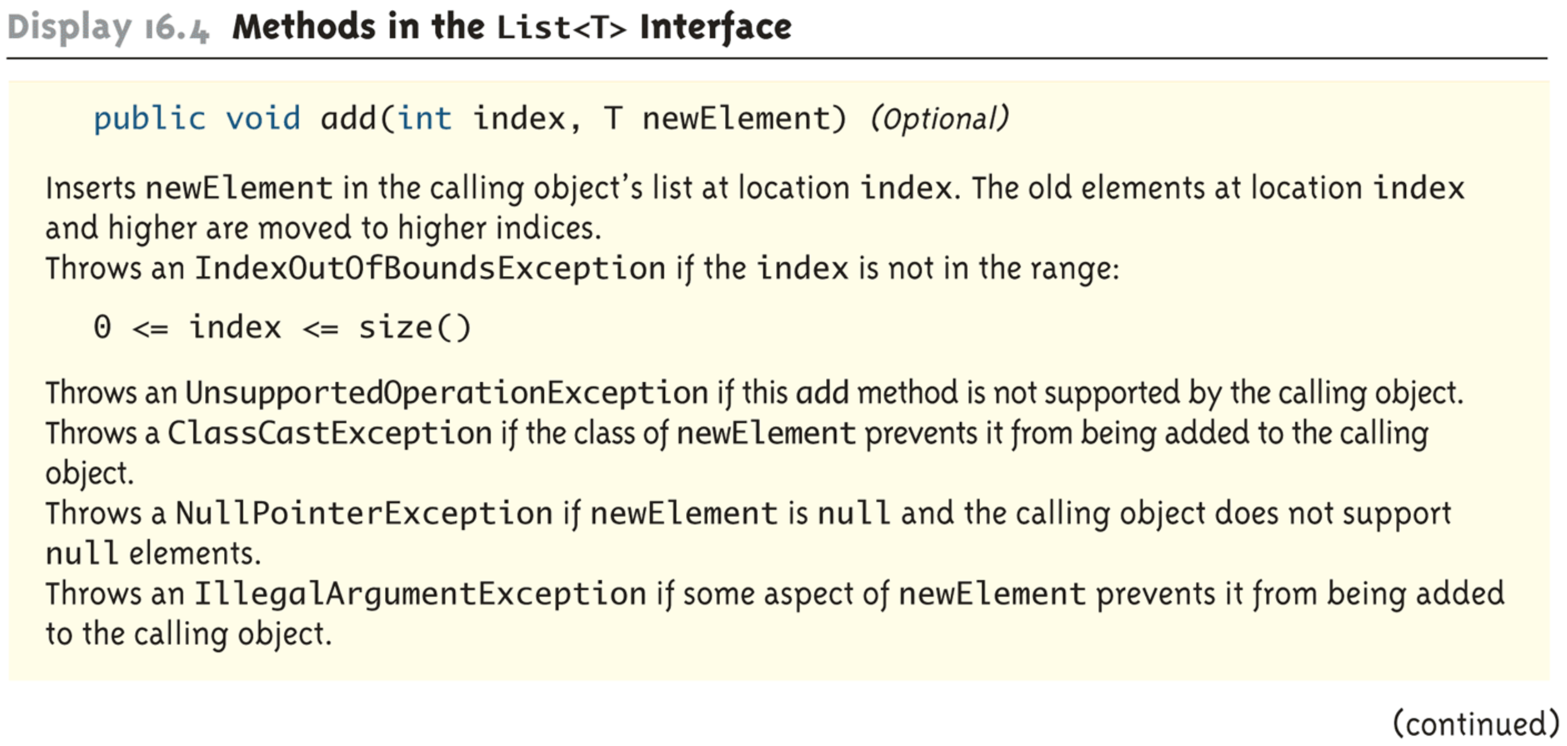
Collection Relationships

- Classes that implement the `List<T>` interface have their elements ordered as on a list
  - Elements are indexed starting with zero
  - A class that implements the `List<T>` interface allows elements to occur more than once
  - The `List<T>` interface has more method headings than the `Collection<T>` interface
  - Some of the methods inherited from the `Collection<T>` interface have different semantics in the `List<T>` interface
  - The `ArrayList<T>` class implements the `List<T>` interface

Some methods in the List<T> Interface

- Semantics for methods defined in Collection<T>
  - equals( ) returns true if the calling object and argument have the same element in the same order
  - toArray( ) returns the copies of the elements (not references) in the same order
  - add( ) places the new element at the “end” of the list

New Methods in the `List<T>` Interface (1 of 6)



Display 16.4 Methods in the `List<T>` Interface

```
public void add(int index, T newElement) (Optional)
```

Inserts `newElement` in the calling object's list at location `index`. The old elements at location `index` and higher are moved to higher indices.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if this add method is not supported by the calling object.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.
Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

(continued)

New Methods in the `List<T>` Interface (Part 2 of 6)

Display 16.4 Methods in the `List<T>` Interface

```
public boolean addAll(int index,
                      Collection<? extends T> collectionToAdd) (Optional)
```

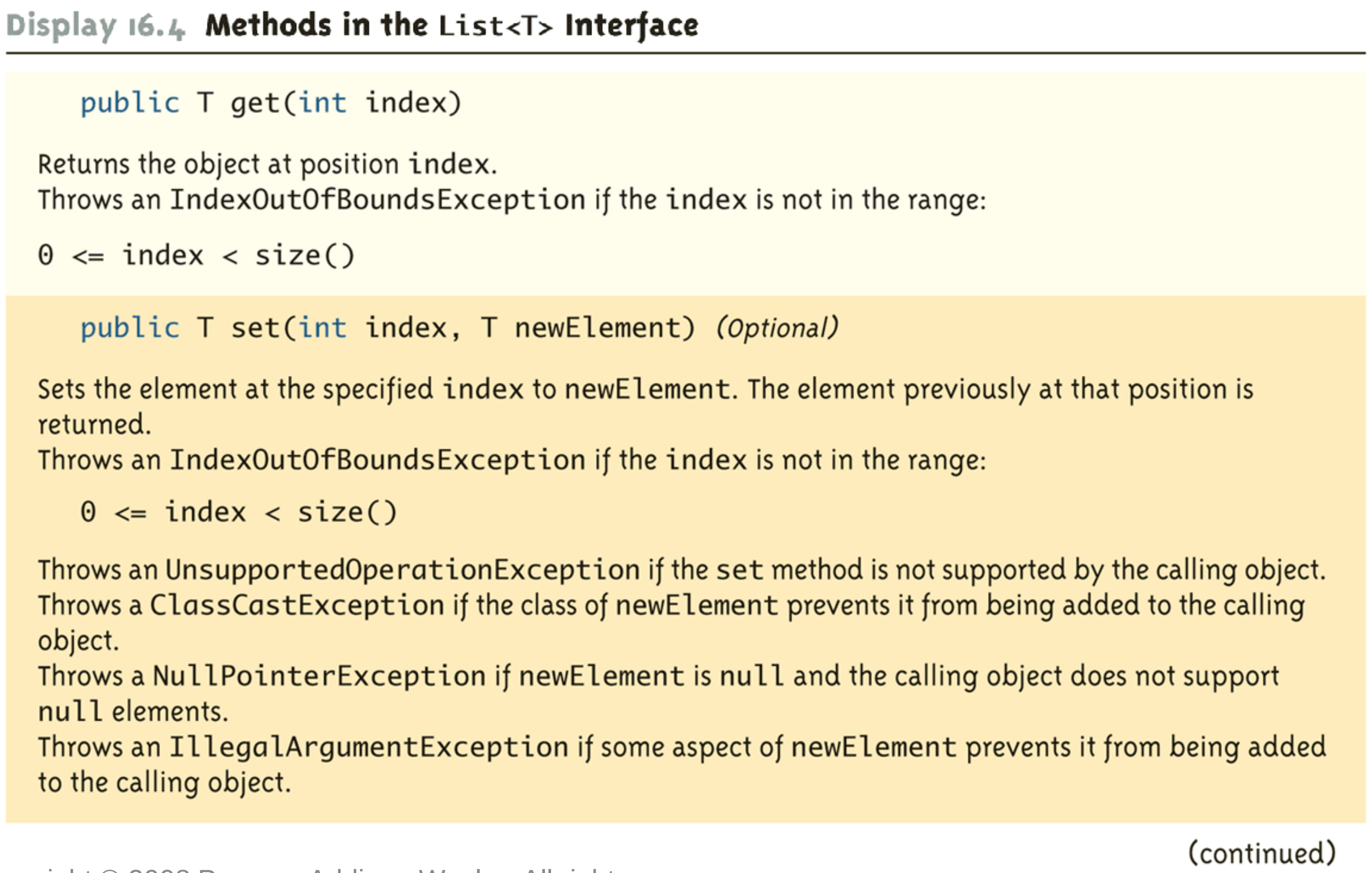
Inserts all of the elements in `collectionToAdd` to the calling object's list starting at location `index`. The old elements at location `index` and higher are moved to higher indices. The elements are added in the order they are produced by an iterator for `collectionToAdd`.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if the `addAll` method is not supported by the calling object.
Throws a `ClassCastException` if the class of one of the elements of `collectionToAdd` prevents it from being added to the calling object.
Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.
Throws an `IllegalArgumentException` if some aspect of one of the elements of `collectionToAdd` prevents it from being added to the calling object.

(continued)

New Methods in the **`List<T>`** Interface (Part 3 of 6)



Display 16.4 Methods in the `List<T>` Interface

```
public T get(int index)
```

Returns the object at position `index`.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index < size()
```

```
public T set(int index, T newElement) (Optional)
```

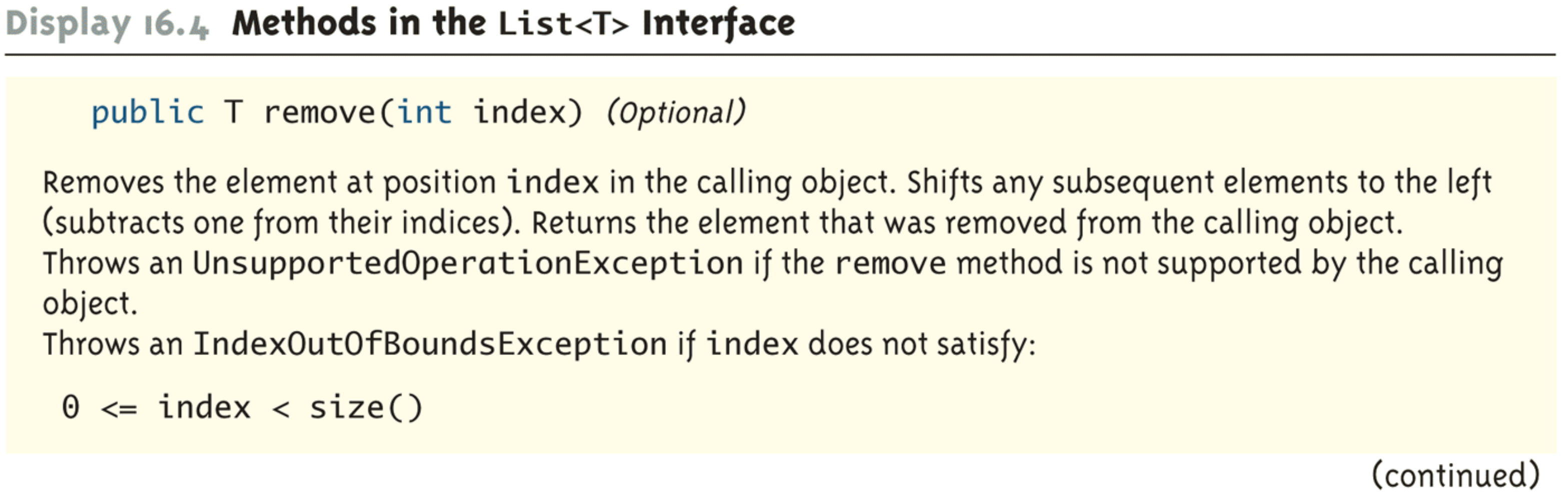
Sets the element at the specified `index` to `newElement`. The element previously at that position is returned.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index < size()
```

Throws an `UnsupportedOperationException` if the `set` method is not supported by the calling object.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.
Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

(continued)

New Methods in the `List<T>` Interface (Part 4 of 6)



Display 16.4 Methods in the `List<T>` Interface

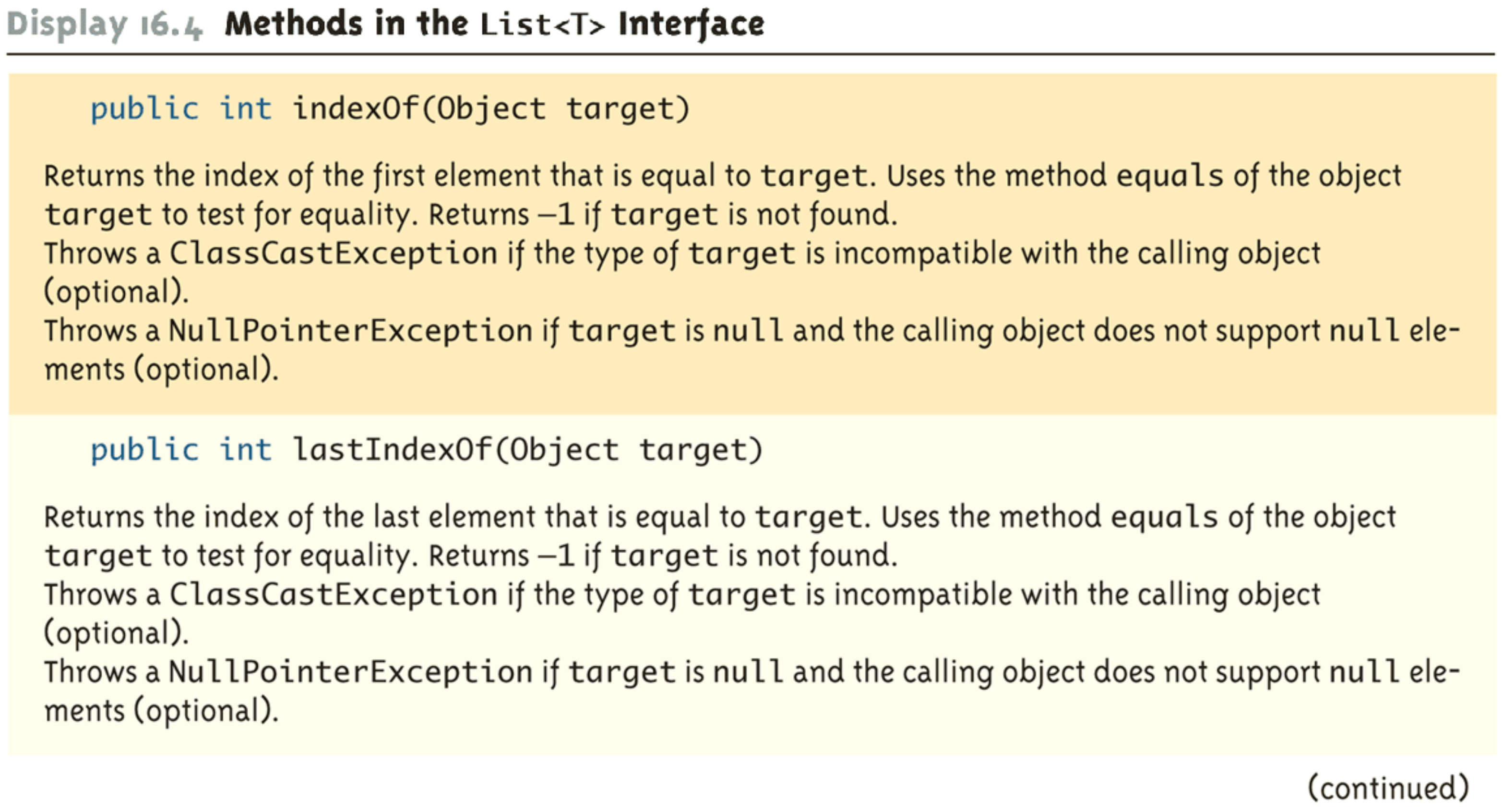
```
public T remove(int index) (Optional)
```

Removes the element at position `index` in the calling object. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the calling object.
Throws an `UnsupportedOperationException` if the `remove` method is not supported by the calling object.
Throws an `IndexOutOfBoundsException` if `index` does not satisfy:

```
0 <= index < size()
```

(continued)

New Methods in the `List<T>` Interface (Part 5 of 6)



Display 16.4 Methods in the `List<T>` Interface

```
public int indexOf(Object target)
```

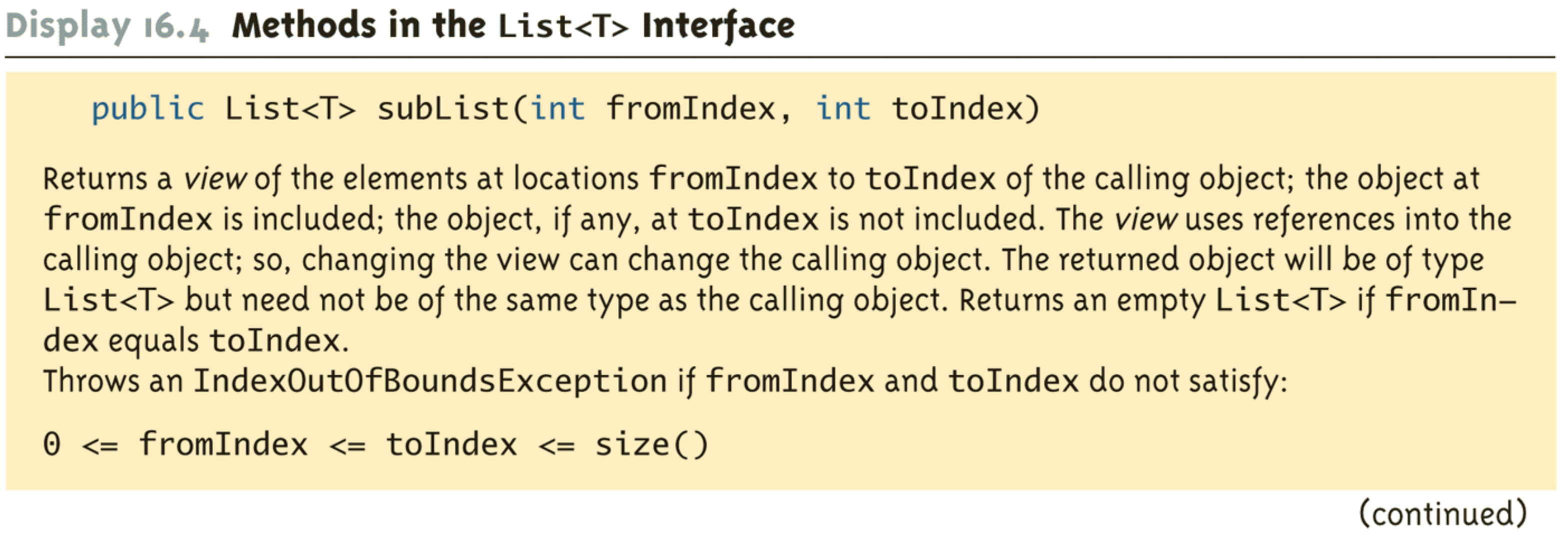
Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns −1 if `target` is not found.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns −1 if `target` is not found.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

(continued)

New Methods in the `List<T>` Interface (Part 6 of 6)



Display 16.4 Methods in the `List<T>` Interface

```
public List<T> subList(int fromIndex, int toIndex)
```

Returns a *view* of the elements at locations `fromIndex` to `toIndex` of the calling object; the object at `fromIndex` is included; the object, if any, at `toIndex` is not included. The *view* uses references into the calling object; so, changing the view can change the calling object. The returned object will be of type `List<T>` but need not be of the same type as the calling object. Returns an empty `List<T>` if `fromIn-dex` equals `toIndex`.
Throws an `IndexOutOfBoundsException` if `fromIndex` and `toIndex` do not satisfy:

```
0 <= fromIndex <= toIndex <= size()
```

(continued)

Classes that implement List<T>

- If you do not need any methods beyond those defined in List<T>, but need a List that provides fast random access to the elements (the get( ) method), use ArrayList<T> (or the older Vector<T>).
  - But note that inserting or deleting from the middle of the ArrayList or Vector will be slow
- If you do not need fast random access, but require efficient sequential access through the list, use LinkedList<T>
  - Inserting or deleting from the middle of the LinkedList is faster than with ArrayList or Vector

List<T> example code

```
public class ListExample {
   public static void main( String[ ] args)
  {
        // Note the use of List<Integer> here
        List<Integer> list = new ArrayList<Integer>( );

        // add elements to the end of the list
        for (int k = 0; k < 10; k++)
              list.add(k*2);              // autoboxing
        for (Integer k : list)
              System.out.print( k + ", ");
        System.out.println( );

        // remove element at index 4
        list.remove( 4 );
        for (Integer k : list)
              System.out.print( k + ", ");
        System.out.println( );
```

(continued)

List<T> example code

```
      // insert 99 at index 2
      list.add( 2, 99 );
      for (Integer k : list)
            System.out.print( k + ", ");
      System.out.println( );

      // change the value at index 3 to 77
      list.set( 3, 77 );
      for (Integer k : list)
            System.out.print( k + ", ");
      System.out.println( );
   }
}
// Output ---
0, 2, 4, 6, 8, 10, 12, 14, 16, 18,
0, 2, 4, 6, 10, 12, 14, 16, 18,
0, 2, 99, 4, 6, 10, 12, 14, 16, 18,
0, 2, 99, 77, 6, 10, 12, 14, 16, 18,
```

The Collections Class

In addition to the Collection<T> interface, Java provides the Collections class that contains static methods to operate on or return collections.

Some method of this class related to List are

`static void reverse( List<?> list)`

`that reverses the contents of the specified list.`

`static <T extends Comparable<? super T>`
`void sort( List<T> list)`

`that sorts the specified list`

`static void <T>`
`copy( List<? super T> dest, List<? extends T> source)`

`copies source List to destination List`

Collection Relationships

- Classes that implement the **`Set<T>`** interface do not allow an element in the class to occur more than once
  - The **`Set<T>`** interface has the same method headings as the **`Collection<T>`** interface, but in some cases the *semantics* (intended meanings) are different
  - Methods that are optional in the **`Collection<T>`** interface are required in the **`Set<T>`** interface

Methods in the `Set<T>` interface

- The `Set<T>` interface has the same method headings as the `Collection<T>` interface, but in some cases the semantics are different. For example the `add` methods:

`public boolean add(T element)` ***(Optional)***

If element is not already in the calling object, element is added to the calling object and true is returned. If element is in the calling object, the calling object is unchanged and false is returned.

`public boolean addAll(Collection<? extends T> collectionToAdd)` ***(Optional)***

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns true if the calling object changed as a result of the call; returns false otherwise. Thus, if `collectionToAdd` is a `Set<T>`, then the calling object is changed to the union of itself with `collectionToAdd`.

Classes that implement Set<T>

- All classes that implement Set<T> disallow duplicate elements
- If you just need a collection that does not allow duplicates, use HashSet<T>
- If you also need fast sorted element access, use TreeSet<T>

More Collections Methods

Some methods of the Collections class related to Sets include

```
static <T> Set<T> singleton(T obj)
   that returns an immutable set containing only the
   specified object

static <T> Set<T> emptySet( )
   that returns an (immutable) empty set
```

Iterator Concept

- For any container, there must be a way to insert elements and retrieve them again.
- If we write code specific for the container we use in our application, then that code must be changed if we decide to change the container we use. A better approach is to write more general-purpose code that works with any container.
- An **Iterator** is an object that allows us to write this general purpose code.
  - An iterator object allows us to sequentially access the elements of a container without knowing or caring how the elements are held in the container.

The `Iterator<T>` Interface

- Java provides an `Iterator<T>` interface
  - Any object of any class that satisfies the `Iterator<T>` interface is an `Iterator<T>`
- An `Iterator<T>` does not stand on its own
  - It must be associated with some collection object using the method `iterator`
  - If `c` is an instance of a collection class (e.g., `HashSet<String>`), the following obtains an iterator for `c`. The collection creates the iterator.

```
Iterator<String> iteratorForC = c.iterator();
```

Methods in the Iterator<T> Interface

- public T next( ) – returns the next element of the collection.
- public boolean hasNext( ) – returns true if the collection has more elements. False otherwise.
- public void remove( ) – removes the last element returned by next( ) (optional)

Using an Iterator with a HashSet<T> Object

- A **`HashSet<T>`** object imposes no order on the elements it contains
- However, an iterator will impose an order on the elements in the hash set
  - That is, the order in which they are produced by **`next()`**
  - Although the order of the elements so produced may be duplicated for each program run, there is no requirement that this must be the case

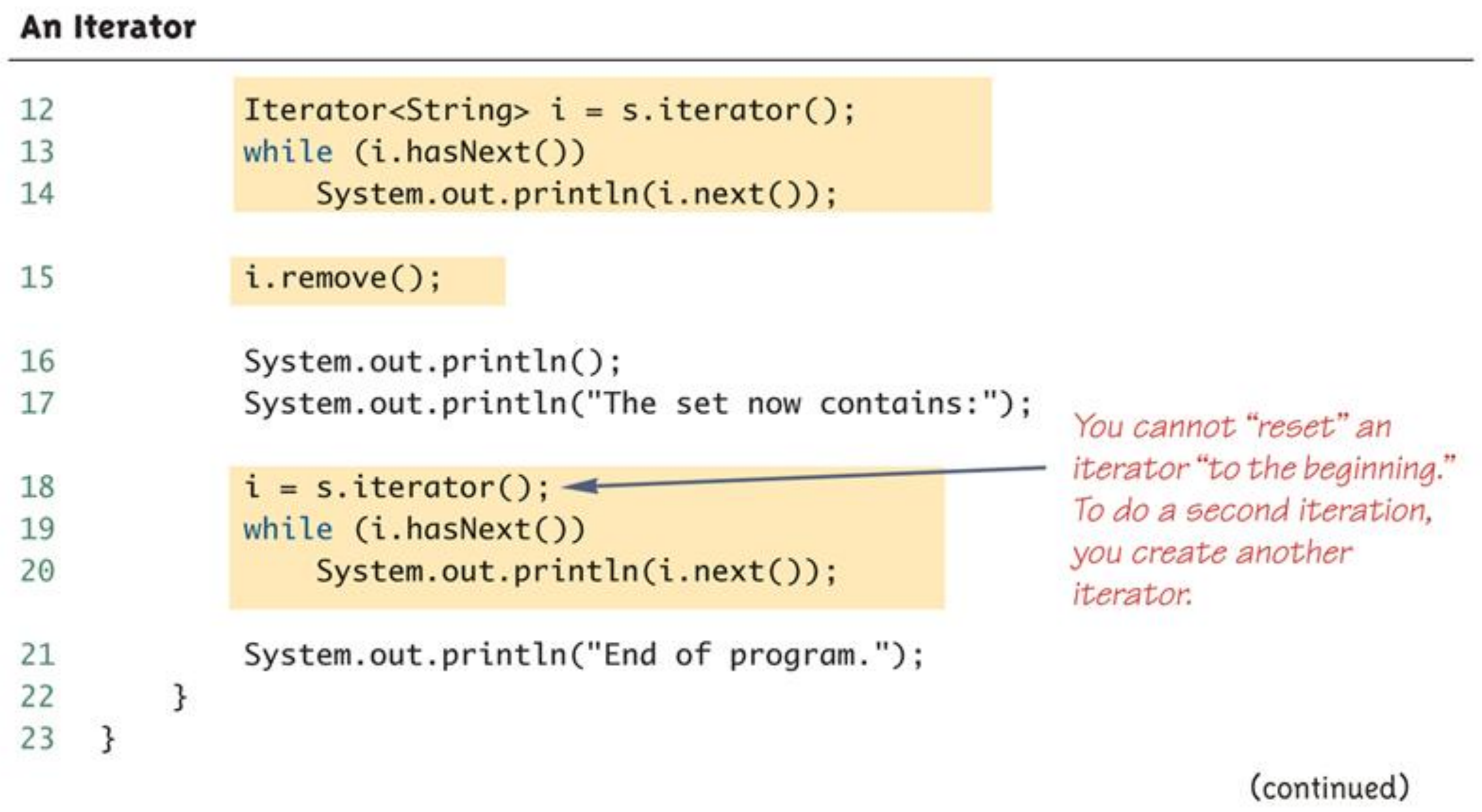
An Iterator Example (1 of 3)

An Iterator

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetIteratorDemo
{
    public static void main(String[] args)
    {
        HashSet<String> s = new HashSet<String>();

        s.add("health");
        s.add("love");
        s.add("money");

        System.out.println("The set contains:");
```

(continued)

An Iterator Example (2 of 3)



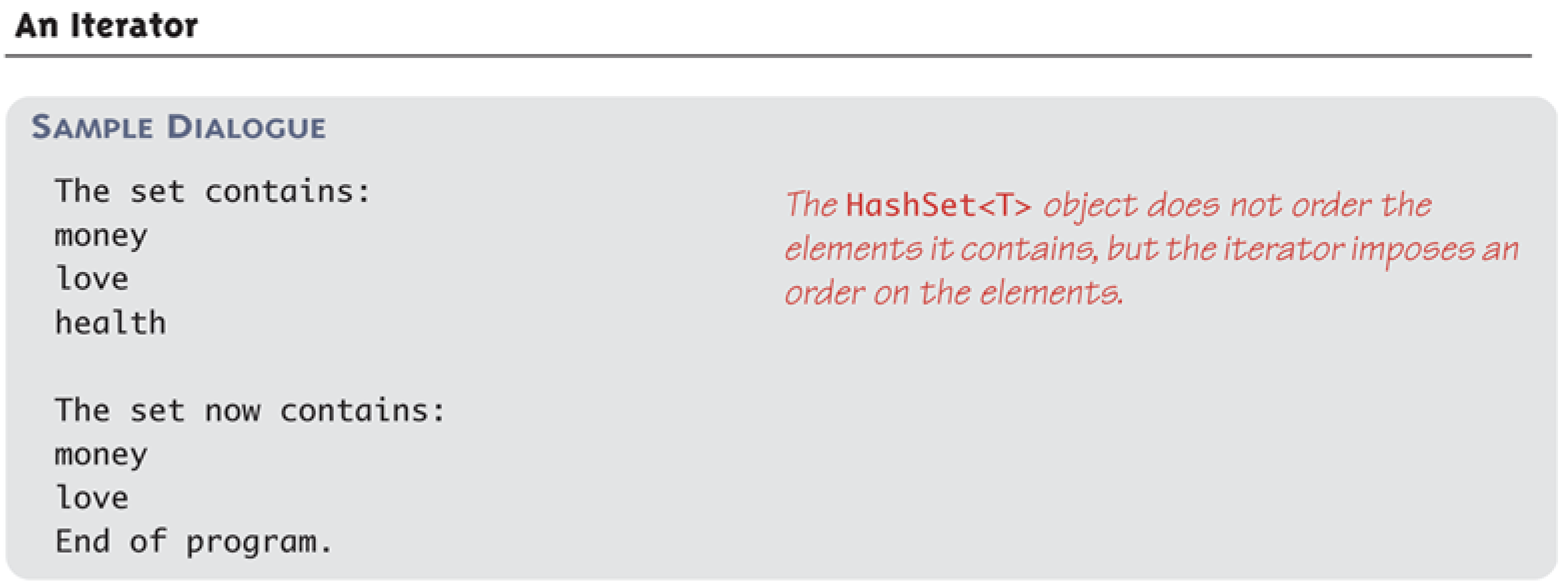
An Iterator

```
    Iterator<String> i = s.iterator();
    while (i.hasNext())
      System.out.println(i.next());

    i.remove();

    System.out.println();
    System.out.println("The set now contains:");

    i = s.iterator();
    while (i.hasNext())
      System.out.println(i.next());

    System.out.println("End of program.");
  }
}
```

You cannot "reset" an iterator "to the beginning." To do a second iteration, you create another iterator.

(continued)

An Iterator Example (3 of 3)



An Iterator

SAMPLE DIALOGUE

```
The set contains:
money
love
health

The set now contains:
money
love
End of program.
```

The `HashSet<T>` object does not order the elements it contains, but the iterator imposes an order on the elements.

For-Each vs. Iterator

- How is using an iterator any different from using a for-each loop (which seems easier)?
  - Both for-each and iterators work with any collection. If you’re just walking through the elements, for-each is more succinct.
  - An iterator can be used to remove elements
  - You can create as many (independent) iterators for the same collection simultaneously as you need

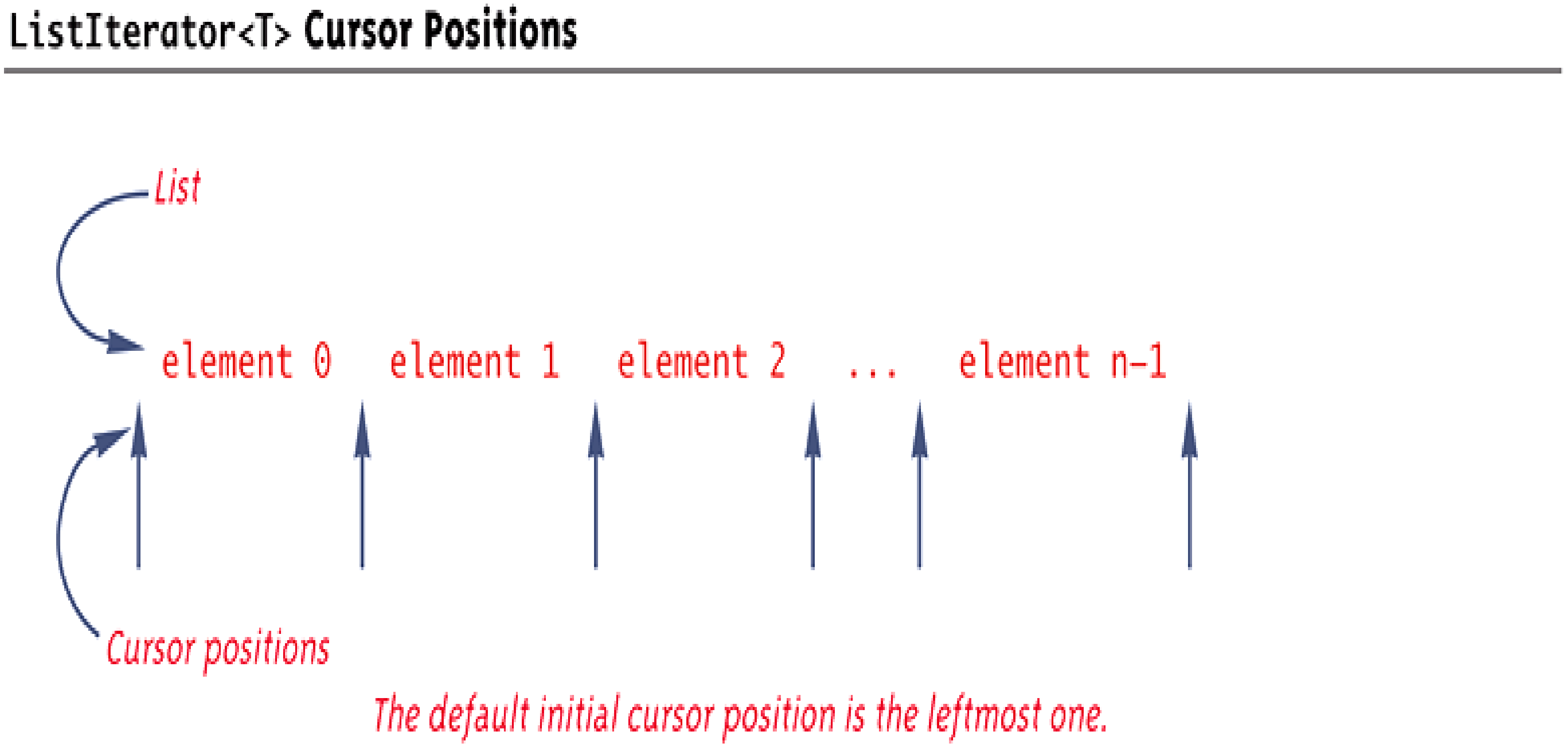
The `ListIterator<T>` Interface

- The `ListIterator<T>` interface extends the `Iterator<T>` interface, and is designed to work with collections that satisfy the `List<T>` interface
  - A `ListIterator<T>` has all the methods that an `Iterator<T>` has, plus additional methods
  - A `ListIterator<T>` can move in either direction along a list of elements
  - A `ListIterator<T>` has methods, such as `set` and `add`, that can be used to modify elements

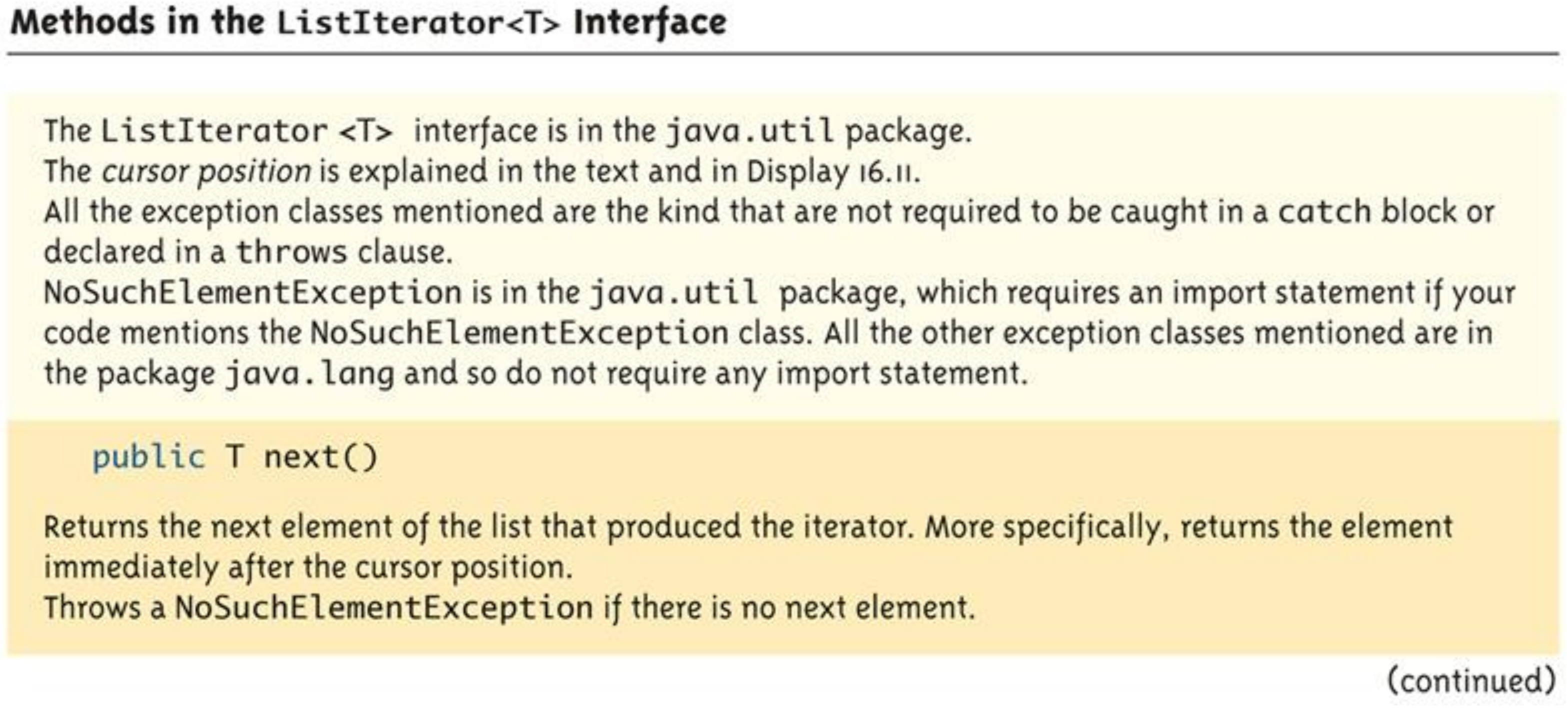
The **`ListIterator<T>`** Cursor

- Every **`ListIterator<T>`** has a position marker known as the *cursor*
  - If the list has *n* elements, they are numbered by indices 0 through *n*-1, but there are *n*+1 cursor positions
  - When **`next()`** is invoked, the element immediately following the cursor position is returned and the cursor is moved forward one cursor position
  - When **`previous()`** is invoked, the element immediately before the cursor position is returned and the cursor is moved back one cursor position

`ListIterator<T>` Cursor Positions


ListIterator<T> Cursor Positions
List
element 0
element 1
element 2
...
element n-1
Cursor positions
The default initial cursor position is the leftmost one.

Methods in the `ListIterator<T>` Interface (Part 1 of 4)



Methods in the `ListIterator<T>` Interface

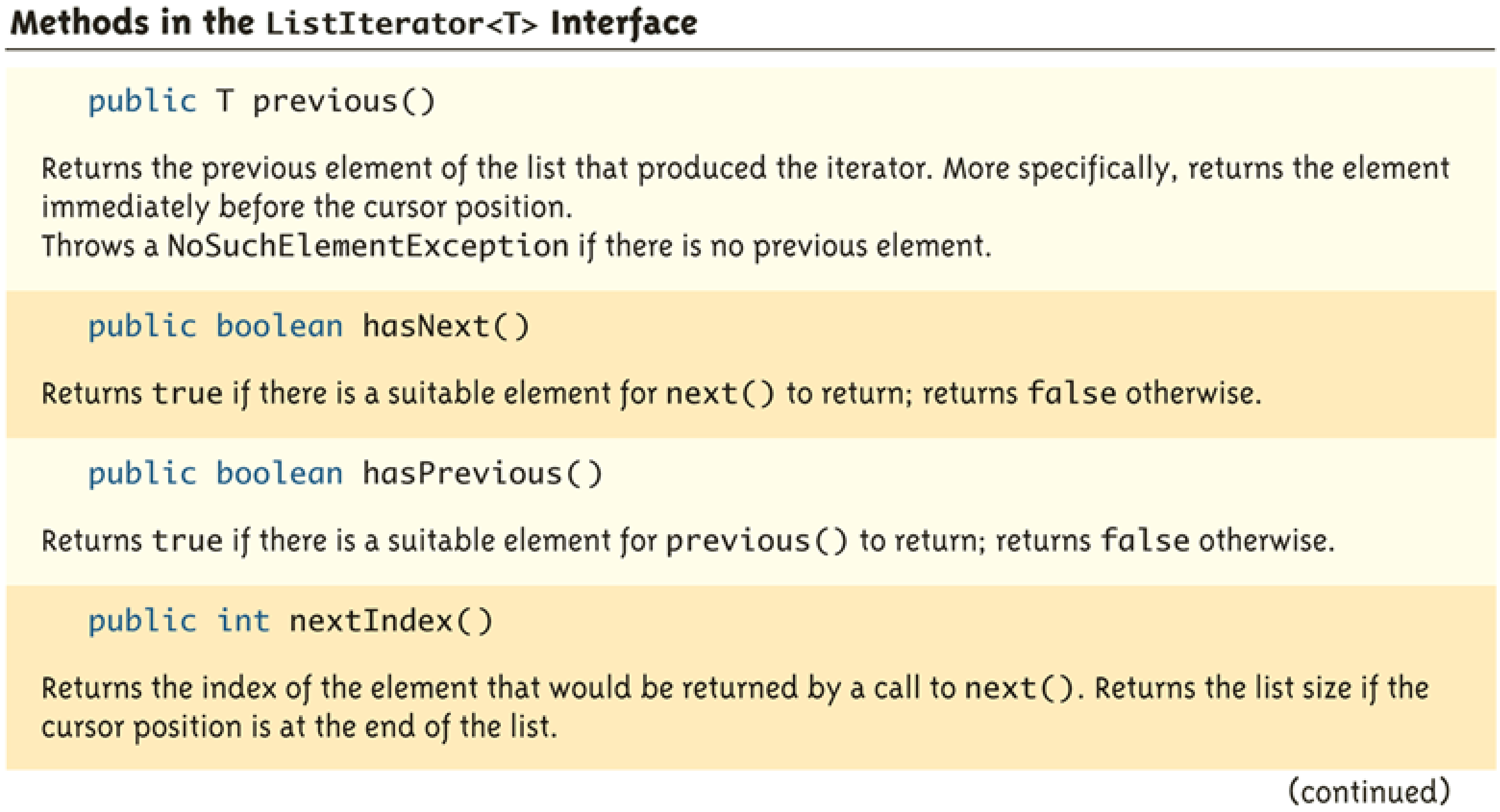
The `ListIterator <T>` interface is in the `java.util` package.
The *cursor position* is explained in the text and in Display 16.11.
All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.
`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public T next()
```

Returns the next element of the list that produced the iterator. More specifically, returns the element immediately after the cursor position.
Throws a `NoSuchElementException` if there is no next element.

(continued)

Methods in the `ListIterator<T>` Interface (Part 2 of 4)



Methods in the `ListIterator<T>` Interface

```
public T previous()
```

Returns the previous element of the list that produced the iterator. More specifically, returns the element immediately before the cursor position.
Throws a `NoSuchElementException` if there is no previous element.

```
public boolean hasNext()
```

Returns `true` if there is a suitable element for `next()` to return; returns `false` otherwise.

```
public boolean hasPrevious()
```

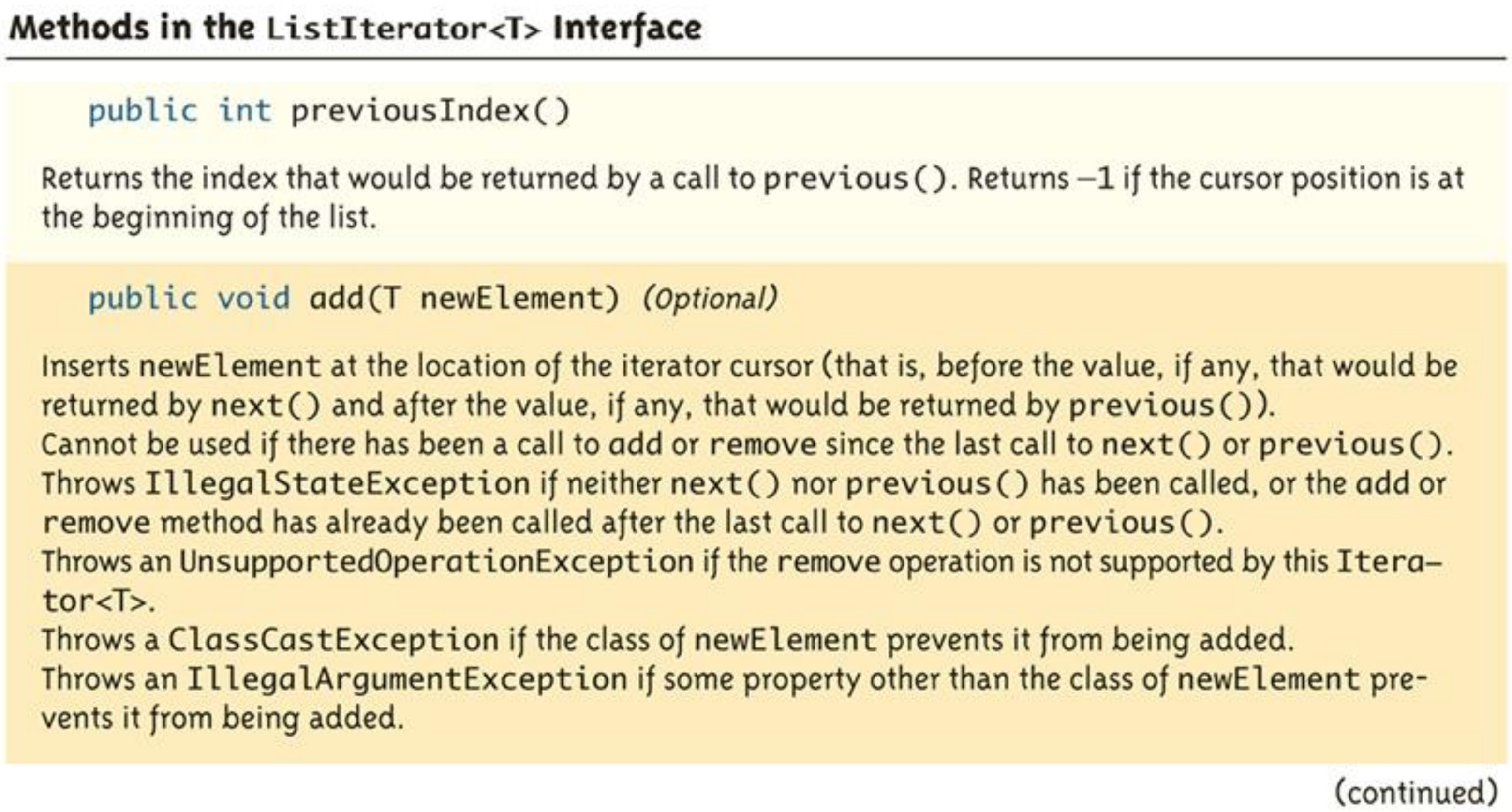
Returns `true` if there is a suitable element for `previous()` to return; returns `false` otherwise.

```
public int nextIndex()
```

Returns the index of the element that would be returned by a call to `next()`. Returns the list size if the cursor position is at the end of the list.

(continued)

Methods in the `ListIterator<T>` Interface (Part 3 of 4)



Methods in the `ListIterator<T>` Interface

`public int previousIndex()`

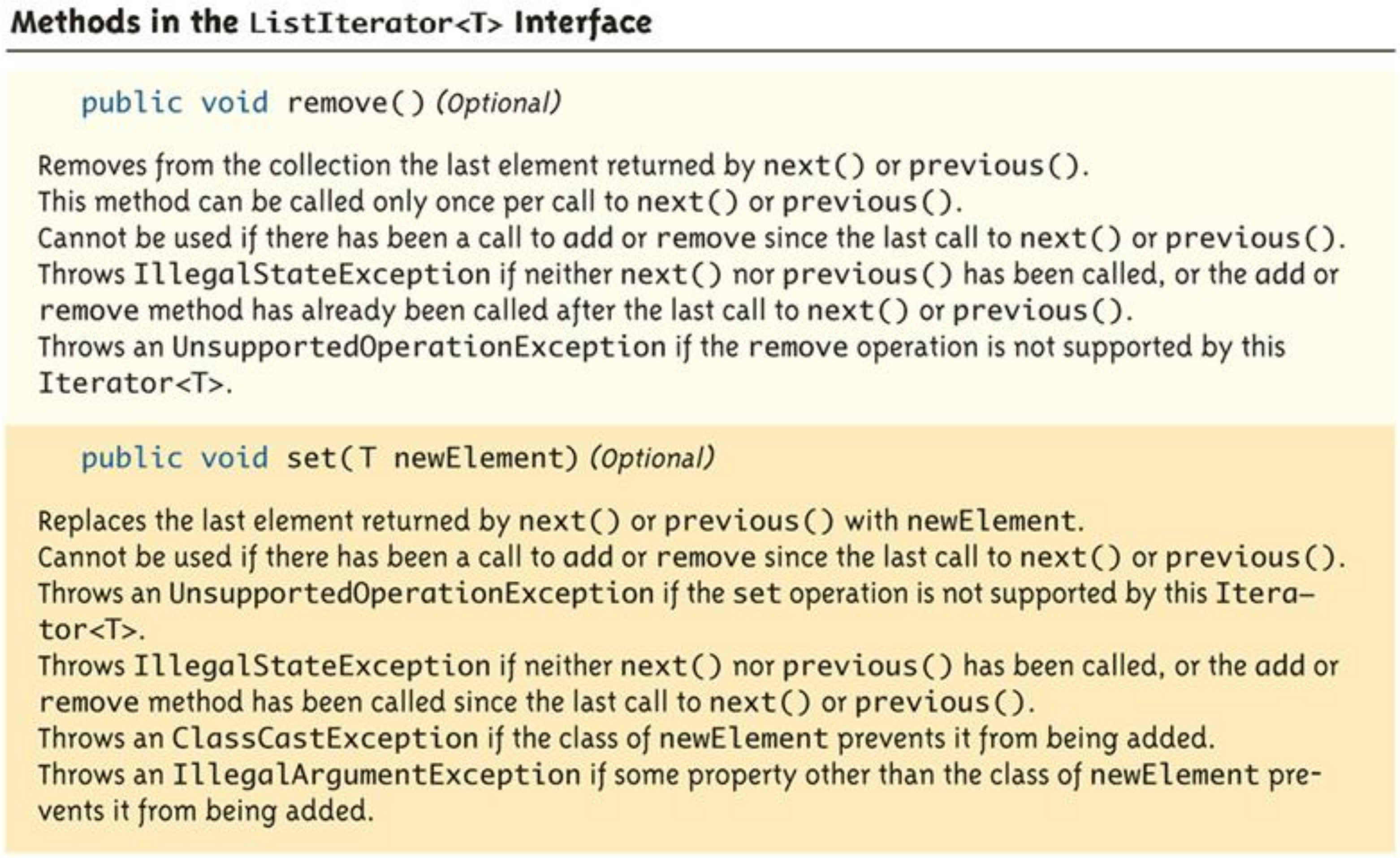
Returns the index that would be returned by a call to `previous()`. Returns −1 if the cursor position is at the beginning of the list.

`public void add(T newElement)` *(Optional)*

Inserts `newElement` at the location of the iterator cursor (that is, before the value, if any, that would be returned by `next()` and after the value, if any, that would be returned by `previous()`).
Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.
Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.
Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Itera-tor<T>`.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added.
Throws an `IllegalArgumentException` if some property other than the class of `newElement` prevents it from being added.

(continued)

Methods in the **`ListIterator<T>`** Interface (Part 4 of 4)



Methods in the `ListIterator<T>` Interface

`public void remove()` *(Optional)*

Removes from the collection the last element returned by `next()` or `previous()`.
This method can be called only once per call to `next()` or `previous()`.
Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.
Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.
Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator<T>`.

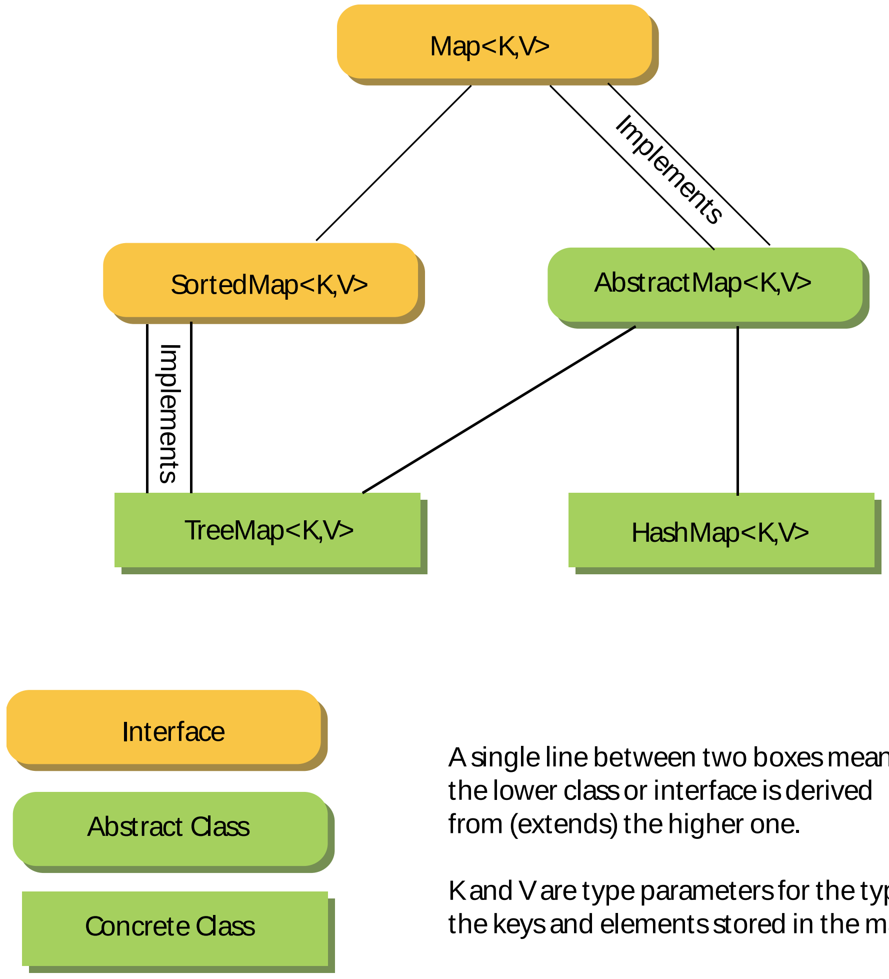
`public void set(T newElement)` *(Optional)*

Replaces the last element returned by `next()` or `previous()` with `newElement`.
Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.
Throws an `UnsupportedOperationException` if the `set` operation is not supported by this `Itera-tor<T>`.
Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has been called since the last call to `next()` or `previous()`.
Throws an `ClassCastException` if the class of `newElement` prevents it from being added.
Throws an `IllegalArgumentException` if some property other than the class of `newElement` pre-vents it from being added.

The Map Framework

- The Java *map* framework deals with collections of ordered pairs
  - For example, a key and an associated value
- Objects in the map framework can implement mathematical functions and relations, so can be used to construct database classes
- The map framework uses the **`Map<T>`** interface, the **`AbstractMap<T>`** class, and classes derived from the **`AbstractMap<T>`** class

The Map Landscape



A single line between two boxes means the lower class or interface is derived from (extends) the higher one.

K and V are type parameters for the type of the keys and elements stored in the map.

Basic Map<K, V> Interface

- No-argument and copy constructors
- public boolean containsValue(Object value) – returns true if the calling map object contains at least one key that maps to the specified value
- public V get(Object key) – returns the value to which the calling object maps the key. Returns null if the key is not in the map.
- public V put(K key, V value) – associates the key with the value in the map. If the key is already in the map, its old value is replaced by the value argument and returned. Otherwise null is returned. (optional)
- public void putAll(Map<? extends K, ? extends V> toAdd) – adds all mappings from toAdd to the calling map object
- public V remove(Object key) – removes the mapping for the specified key.

Classes that implement Map<K, V>

- If you require rapid access to the value associated with a key, use the HashMap<K, V> class
  - HashMap provides no guarantee as to the order of elements placed in the map.
- If you require the elements to be in sorted order by key, then you should use the TreeMap<K,V>
- If you require the elements to be in insertion order, use the LinkedHashMap<K,V> class

Maps and Iterators

- Maps do not provide the iterator( ) method necessary to create iterators, so it's not possible to iterate through a Map directly.
- Java provides a clever work around.
  - The Map interface provides the methods
    - **Set<Map.Entry<K, V>> entrySet( )** - returns a set view of the map. An iterator can then be used on this set view. Map.Entry is an interface that defines the ordered pair.
    - **Set<K> keySet( )** – returns a set view of all keys in the Map. An iterator can then sequence through this set and each key can be used to retrieve its value
    - **Collection<T> values( )** – returns a collection view of all the values in the map. An iterator can then sequence through the collection.

Map Iterator Example

```
public static void main(String[] args)
{
   Map<String, Integer> map = new HashMap<String, Integer>( );
   map.put( "One", 1);
   map.put("Two", 2);
   map.put( "Three",3);

   // get the set of keys and an iterator
   Set<String> keys = map.keySet( );
   Iterator<String> iter = keys.iterator( );

   // sequence through the keys and get the values
   while (iter.hasNext()){
        String key = iter.next();
        System.out.println(key + ", " + map.get(key));
   }
}
// --- Output --
```

```
Three, 3
One, 1
Two, 2
```