CSSE 220 Day 20

Java Collections Framework

LinkedList Implementation

Work on Markov

CSSE 220 Day 20

- Reminder: Exam #2 is Thursday, Jan 31.
- In order to reduce time pressure, you optionally may take the non-programming part 7:10-7:50 AM.

Answers to your questions

- Abstract Data Types and Data Structures
- Markov
- Material you have read
- Anything else

Today's agenda

- Java Collections Framework
- LinkedList Implementation
- Work on Markov

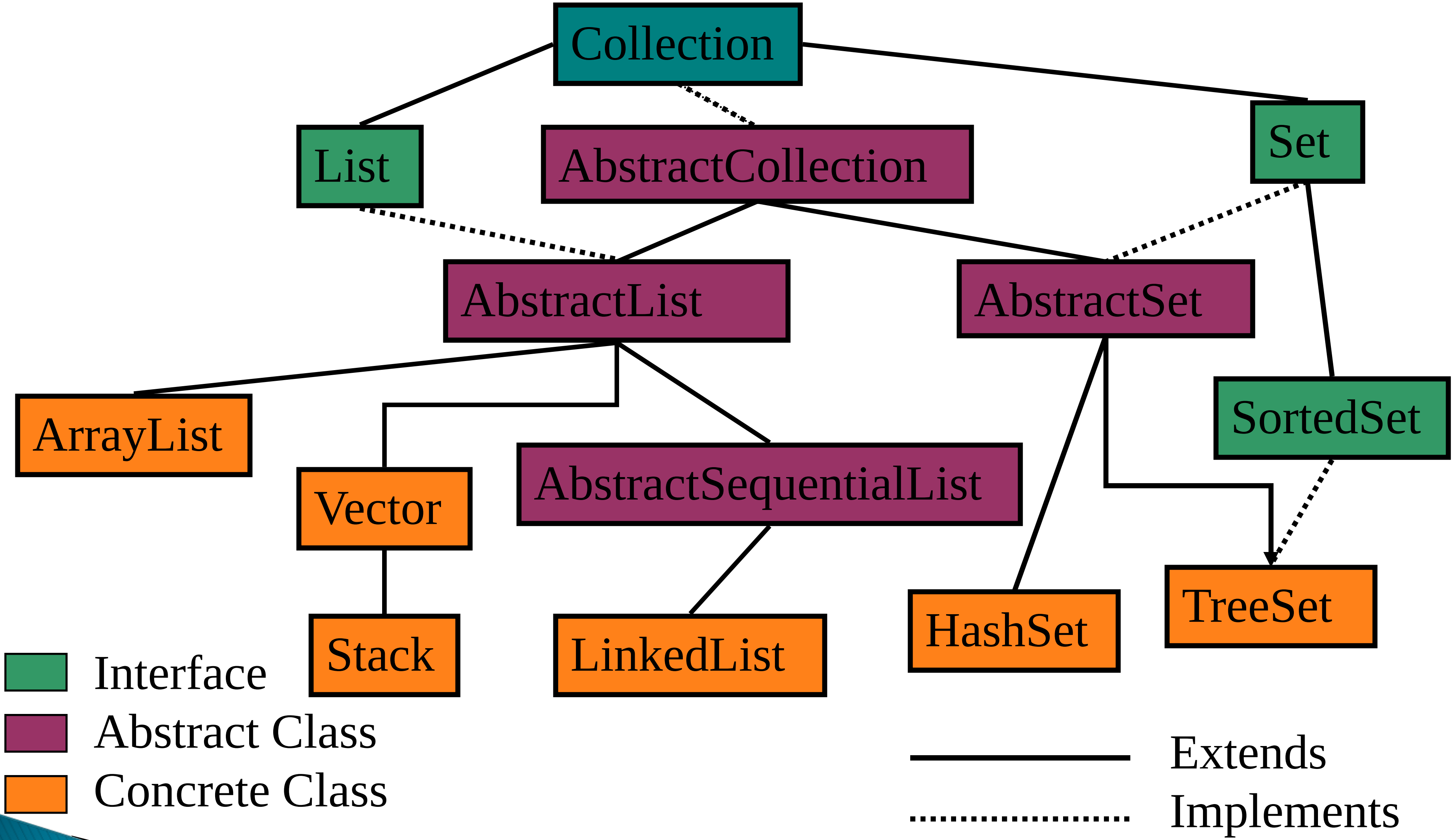
Java Collections Framework Documentation

- Introductory page:
  - http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html
- Outline of the classes:
  - http://java.sun.com/j2se/1.5.0/docs/guide/collections/reference.html
- What’s new in JDK 1.5:
  - http://java.sun.com/j2se/1.5.0/docs/guide/collections/changes5.html

Data Structure Overview

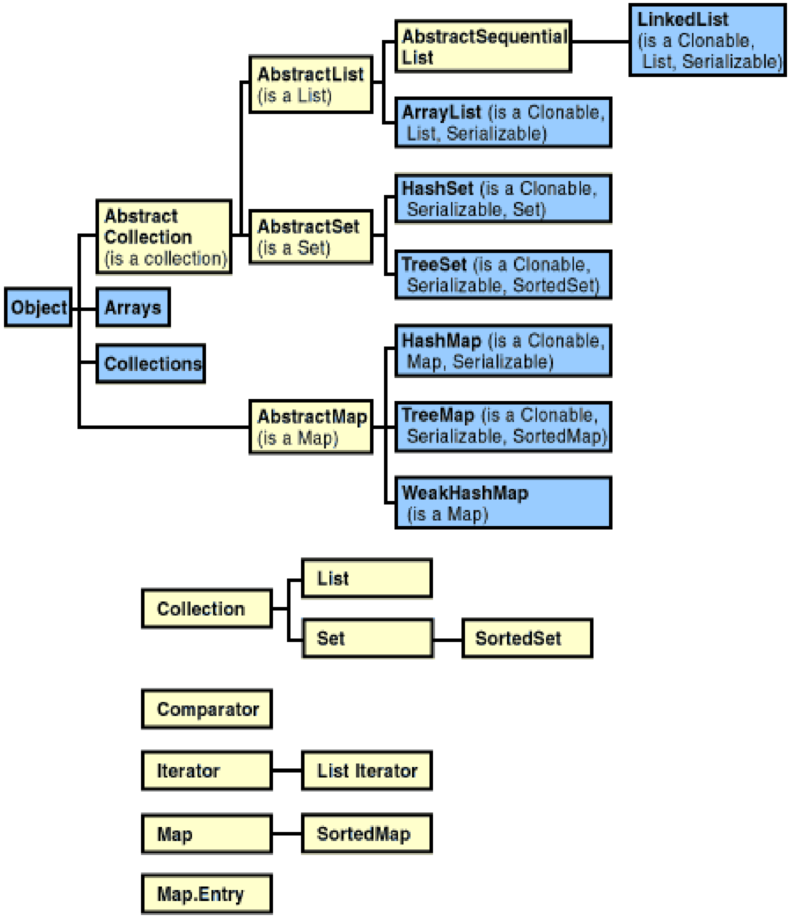
Structure	find	insert/remove	Comments
Array	O(n)	can't do it	Constant-time access by position
Stack	top only O(1)	top only O(1)	Easy to implement as an array.
Queue	front only O(1)	O(1)	insert rear, remove front.
ArrayList	O(log N)	O(N)	Constant-time access by position
Linked List	O(n)	O(1)	O(N) to find insertion position.
HashSet/Map	O(1)	O(1)	If table not too full
TreeSet/Map	O(log N)	O(log N)	Kept in sorted order
MultiSet	O(log N)	O(log N)	keep track of multiplicities
PriorityQueue	O(log N)	O(log N)	Can only find/remove smallest
Tree	O(log N)	O(log N)	If tree is balanced
Graph	O(N*M) ?	O(M)?	N nodes, M edges
Network			shortest path, maxFLow

Some Collection interfaces and classes



This is the Java 1.2 picture. Java 1.5 added Queue, PriorityQueue, and a few other interfaces and classes.

Collections classes and interfaces (classes at top, interfaces at bottom)



Object
Abstract Collection (is a collection)
Arrays
Collections
AbstractList (is a List)
AbstractSet (is a Set)
AbstractMap (is a Map)
AbstractSequential List
ArrayList (is a Clonable, List, Serializable)
LinkedList (is a Clonable, List, Serializable)
HashSet (is a Clonable, Serializable, Set)
TreeSet (is a Clonable, Serializable, SortedSet)
HashMap (is a Clonable, Map, Serializable)
TreeMap (is a Clonable, Serializable, SortedMap)
WeakHashMap (is a Map)
Collection
List
Set
SortedSet
Comparator
Iterator
List Iterator
Map
SortedMap
Map.Entry

Specifying an ADT in Java

- The main Java tool for specifying an ADT is …

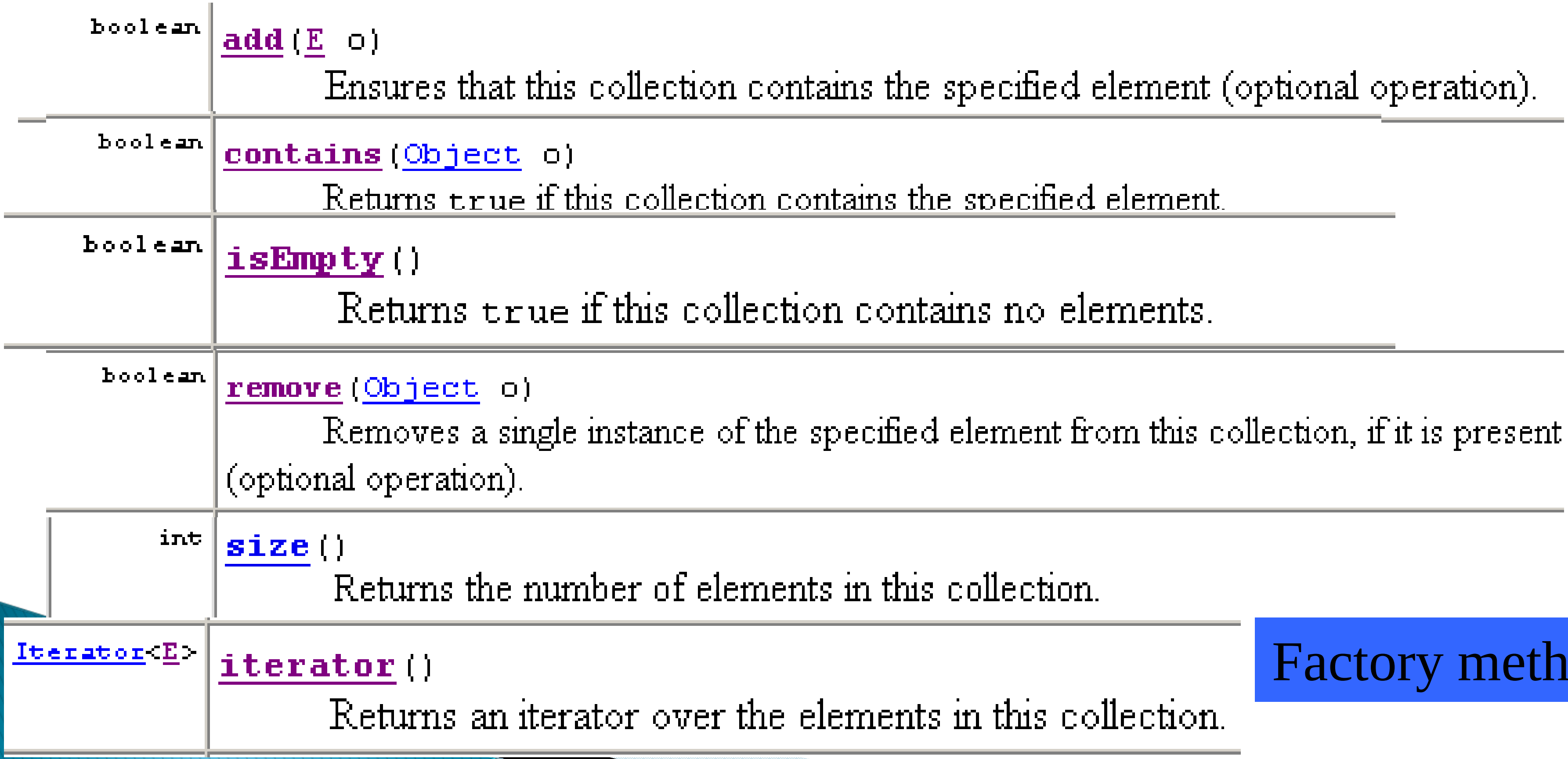
  … an interface

  Major example: The java.util.Collection interface.

java.util

Interface Collection<E>

- Some important methods from this interface:

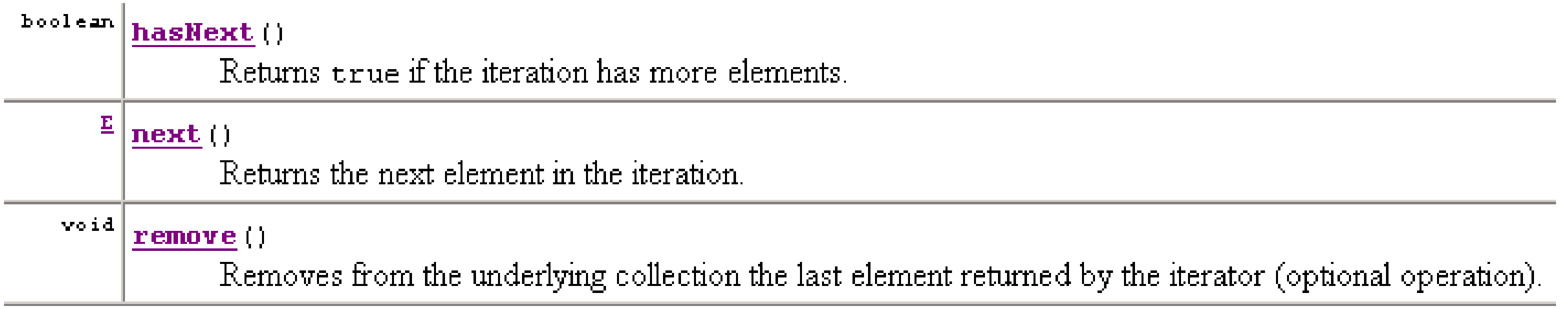


`boolean`	`add(E o)` Ensures that this collection contains the specified element (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this collection contains the specified element.
`boolean`	`isEmpty()` Returns `true` if this collection contains no elements.
`boolean`	`remove(Object o)` Removes a single instance of the specified element from this collection, if it is present (optional operation).
`int`	`size()` Returns the number of elements in this collection.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this collection.

Factory method

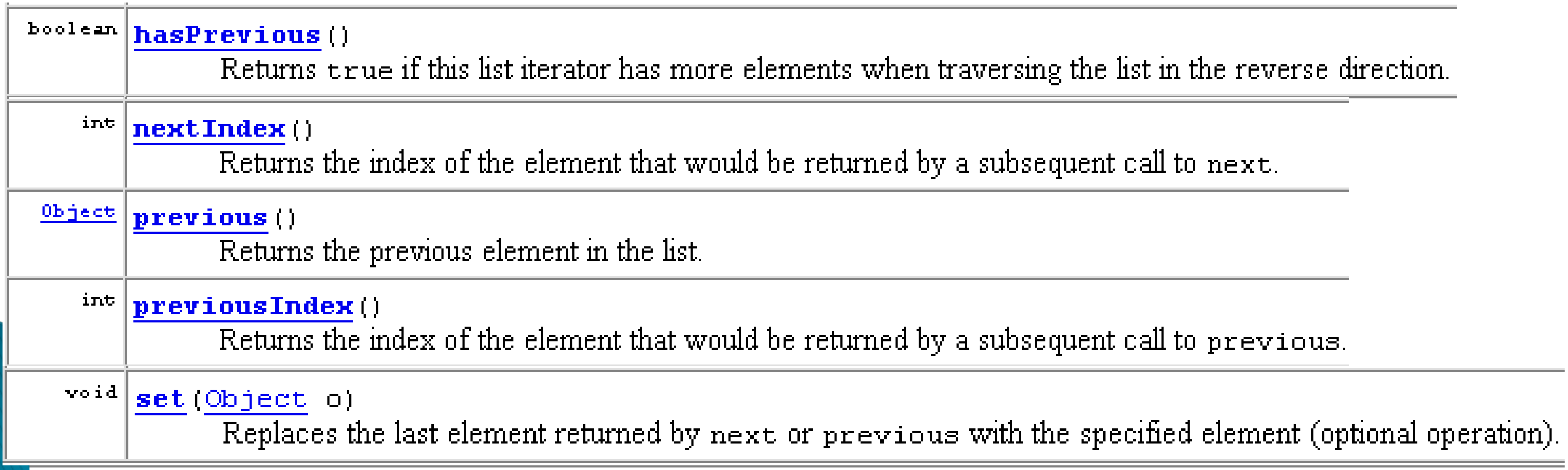
What's an iterator?

- More specifically, what is a `java.util.Iterator`?
  - It's an interface:
  - **`interface java.util.Iterator<E>`**
  - with the following methods:



`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by the iterator (optional operation).

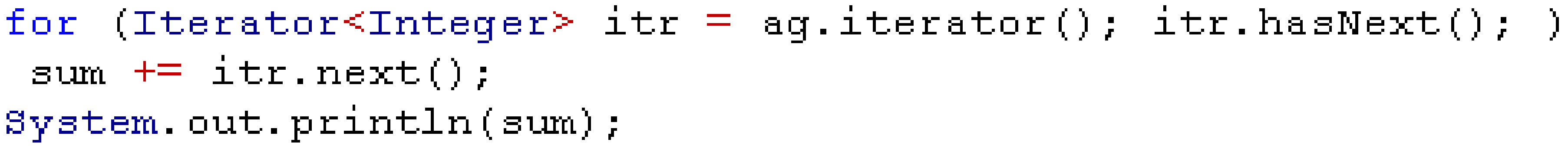
An extension, `ListIterator`, adds:



`boolean`	`hasPrevious()` Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
`int`	`nextIndex()` Returns the index of the element that would be returned by a subsequent call to `next`.
`Object`	`previous()` Returns the previous element in the list.
`int`	`previousIndex()` Returns the index of the element that would be returned by a subsequent call to `previous`.
`void`	`set(Object o)` Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

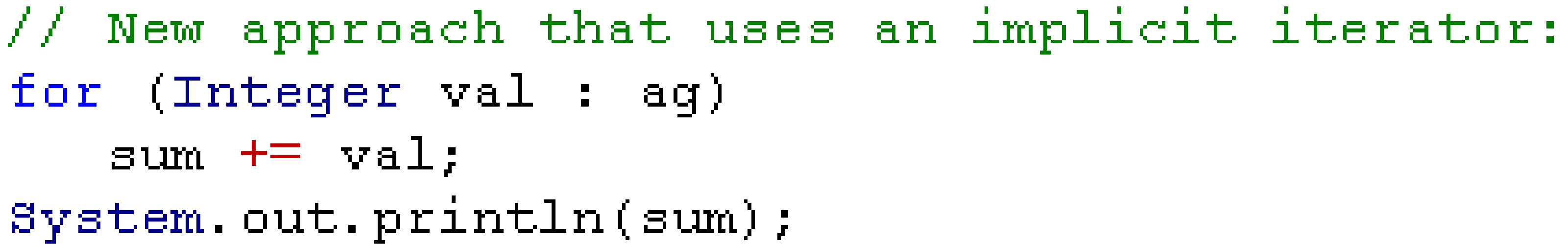
Example: Using an Iterator

In this continuation of the previous example, ag is a Collection object.



```
for (Iterator<Integer> itr = ag.iterator(); itr.hasNext(); )
 sum += itr.next();
System.out.println(sum);
```

In Java 1.5 we can simplify it even more.



```
// New approach that uses an implicit iterator:
for (Integer val : ag)
    sum += val;
System.out.println(sum);
```

Note that the Java compiler translates the latter code into the former.

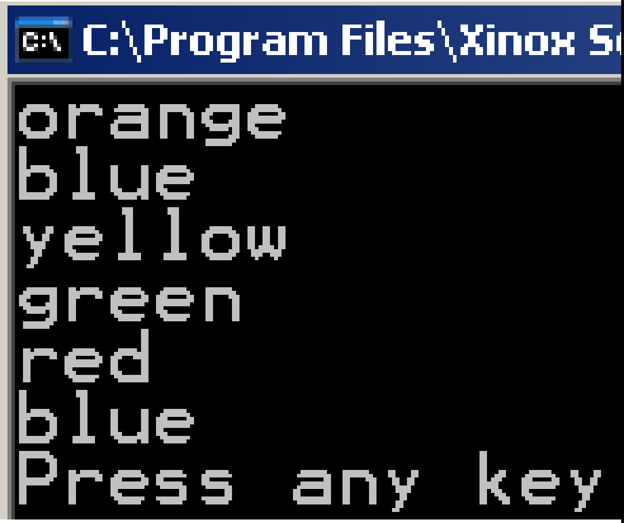
Tangent: Iterating over an enumerated type

```
class EnumTest {
    enum MyColors {orange, blue, yellow, green, red};

    public static void main (String[] args) {
        for (MyColors c : MyColors.values()) {
            System.out.println(c);
        }

        MyColors cc = MyColors.blue;

        switch (cc) {
            case orange:
              System.out.println("It is orange!")
              break;
            case green:
              System.out.println("Oh no! Not green!");
              break;
            case blue:
              System.out.println("blue");
              break;
            default:
              System.out.println("other");
        }
    }
}
```



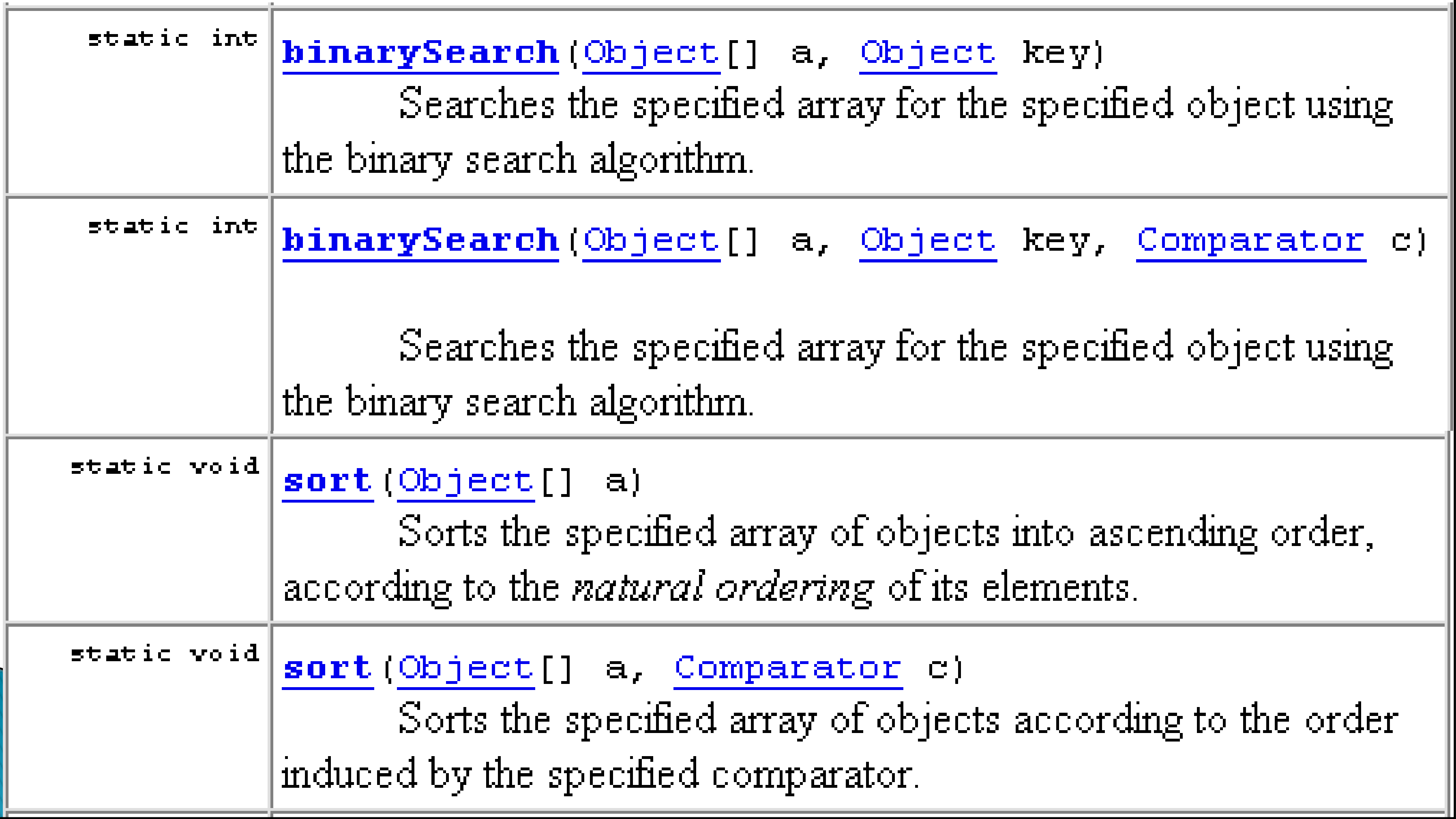
```
orange
blue
yellow
green
red
blue
Press any key
```

Additional Methods from the Collection Interface

- addAll – add all of the elements from another collection to this one
- containsAll – does this collection contain all of the elements of the other collection?
- removeAll – removes all of this collections elements that are also contained in the other collection
- retainAll – removes all of this collections elements that are not contained in the other collection
- toArray – returns an array that contains the same elements as this collection.

Sort and Binary Search

- The `java.util.Arrays` class provides static methods for sorting and doing binary search on arrays. **Examples:**



`static int`	**`binarySearch`**`(Object[] a, Object key)` Searches the specified array for the specified object using the binary search algorithm.
`static int`	**`binarySearch`**`(Object[] a, Object key, Comparator c)` Searches the specified array for the specified object using the binary search algorithm.
`static void`	**`sort`**`(Object[] a)` Sorts the specified array of objects into ascending order, according to the *natural ordering* of its elements.
`static void`	**`sort`**`(Object[] a, Comparator c)` Sorts the specified array of objects according to the order induced by the specified comparator.

Sort and Binary Search

- The **`java.util.Collections`** class provides similar static methods for sorting and doing binary search on `Collections`. Specifically `Lists`.
- **Look up the details in the documentation.**

The weiss.util and weiss.nonstandard packages

- In **weiss.util**, the author shows "bare bones" possible implementations of some of the classes in **java.util**.
- He picks the methods that illustrate the essence of what is involved in the implementation, for educational purposes.
- Some other Data Structures classes are in **weiss.nonstandard**.

The weiss.util and weiss.nonstandard packages

- In **weiss.nonstandard**, the author shows implementations of some common data structures that are not part of the **java.util** package, and he also shows alternate approaches to implementing some classes (like **Stack** and **LinkedList**) that are in **java.util**.

The weiss.util and weiss.nonstandard packages

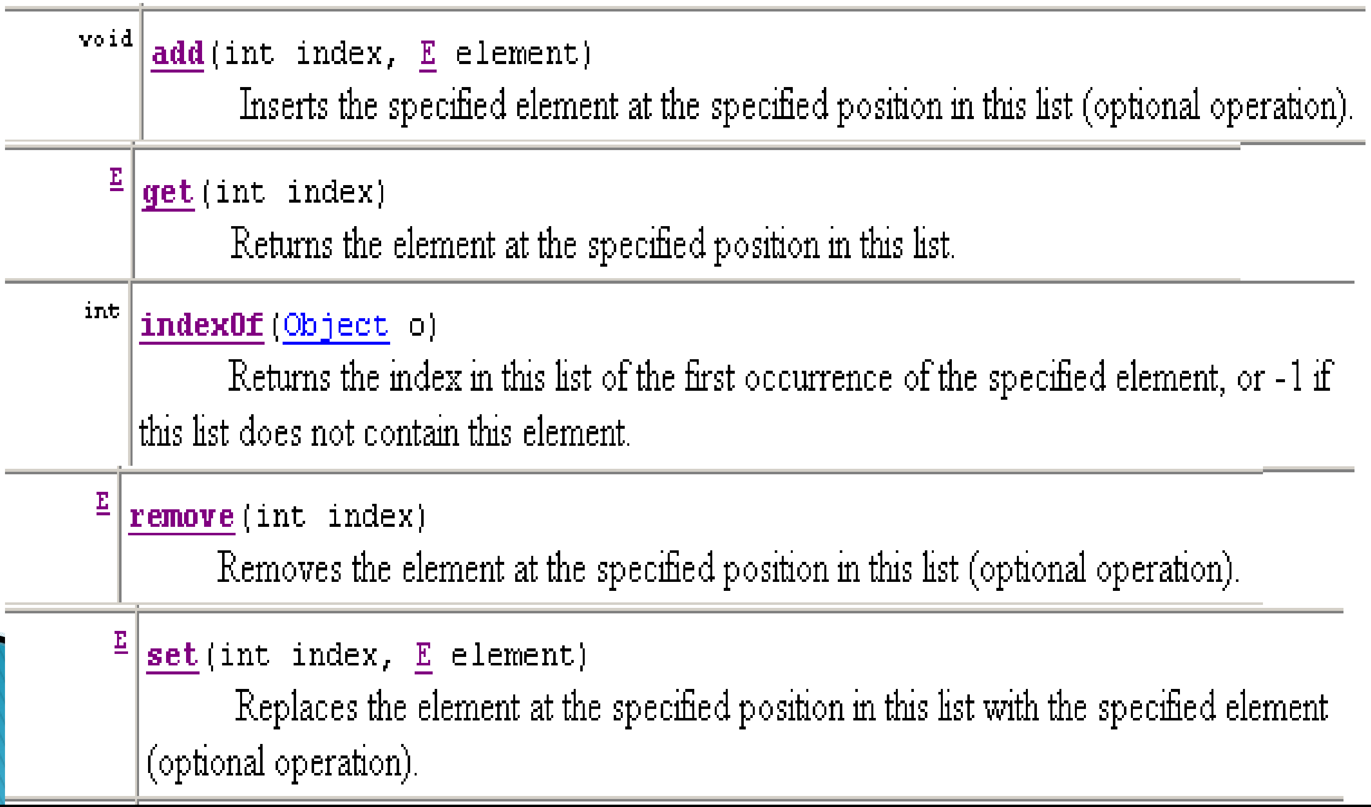
- If you followed the directions in assignment 1, both of these packages should be accessible to your code.
  - import weiss.nonstandard.*;
- Documentation is available, and you can copy it to your computer.

Now that we know about using some data structures …

- It’s time to look at an implementation.

List Interface (extends Collection)

- A List is an ordered collection, items accessible by position. Here, *ordered* does not mean *sorted*.
- interface java.util.List<E>
- User may insert a new item at a specific position.
- Some important List methods:



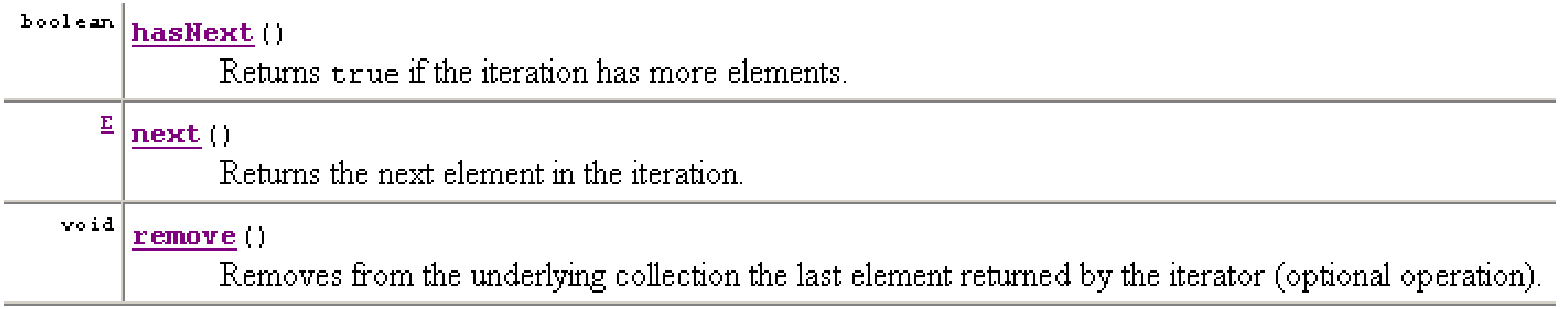
void	add(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
E	get(int index) Returns the element at the specified position in this list.
int	indexOf(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.
E	remove(int index) Removes the element at the specified position in this list (optional operation).
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).

`ArrayList` implementation of the `List` Interface

- Store items contiguously in a "growable" array.
- Looking up an item by index takes constant time.
- Insertion or removal of an object takes linear time in the worst case and on the average (why?).
- If `Comparable` list items are kept in sorted order in the ArrayList, finding an item takes log N time (how?).
- Let's sketch some of the implementation together.
  - Fields, constructor for empty list.

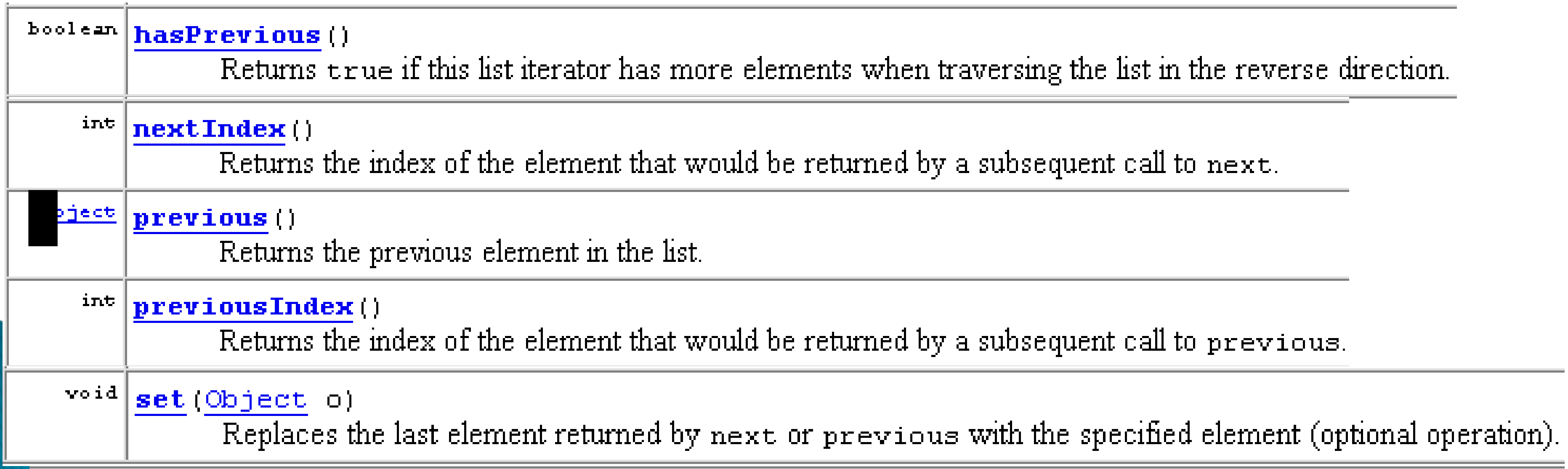
What's an iterator?

- More specifically, what is a `java.util.Iterator`?
  - It's an interface:
  - **`interface java.util.Iterator<E>`**
  - with the following methods:



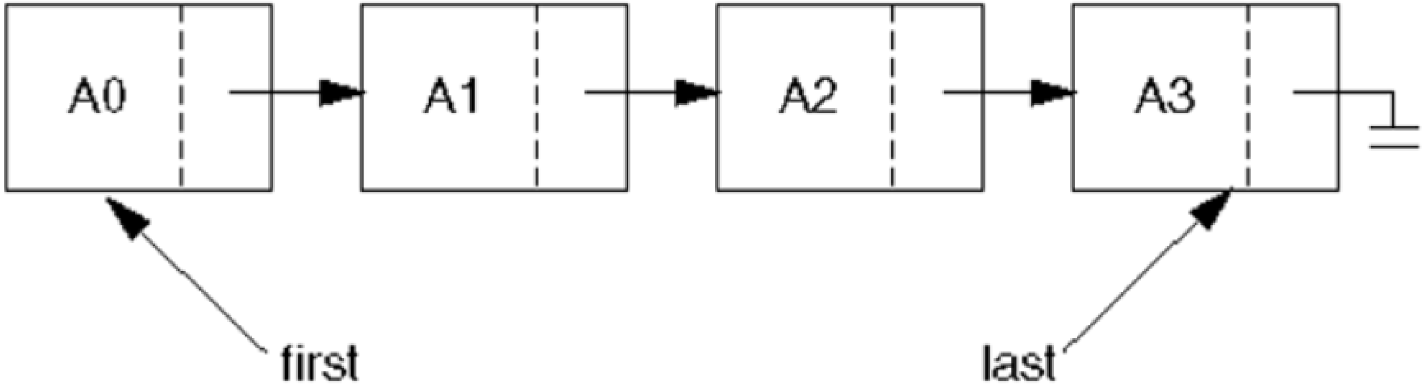
`boolean`	**`hasNext`**`()` Returns `true` if the iteration has more elements.
`E`	**`next`**`()` Returns the next element in the iteration.
`void`	**`remove`**`()` Removes from the underlying collection the last element returned by the iterator (optional operation).

An extension, `ListIterator`, adds:



`boolean`	**`hasPrevious`**`()` Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
`int`	**`nextIndex`**`()` Returns the index of the element that would be returned by a subsequent call to `next`.
[redacted]`ject`	**`previous`**`()` Returns the previous element in the list.
`int`	**`previousIndex`**`()` Returns the index of the element that would be returned by a subsequent call to `previous`.
`void`	**`set`**`(Object o)` Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

LinkedList implementation of the List Interface



- Stores items (non-contiguously) in nodes; each contains a reference to the next node.
- Lookup by index is linear time (worst, average).
- Insertion or removal is constant time once we have found the location.
  - show how to insert A4 after A1.
- If Comparable list items are kept in sorted order, finding an item still takes linear time.

Consider parts of a `LinkedList` implementation

```
class ListNode{
 Object element; // contents of this node
 ListNode next;  // link to next node

 ListNode (Object element,
           ListNode next) {
   this.element = element;
   this.next = next;
 }

 ListNode (Object element) {
   this(element, null);
 }
 ListNode () {
   this(null);
 }
}
```

How to implement LinkedList?

fields?

Constructors?

Methods?

Let's do parts of a `LinkedList` implementation

```
class LinkedList implements List {
  ListNode first;
  ListNode last;
```

Constructors: (a) default (b) single element.

methods:

public **`boolean add(Object o)`**

Appends the specified element to the end of this list (returns `true`)

`public int size()` Returns the number of elements in this list.

`public void add(int i, Object o)` adds o at index i.

```
  throws IndexOutOfBoundsException
```

`public boolean contains(Object o)`

Returns true if this list contains the specified element. (2 versions).

`public boolean remove(Object o)`

Removes the first occurrence (in this list) of the specified element.

`public Iterator iterator()` **Can we also write listIterator( ) ?**

Returns an iterator over the elements in this list in proper sequence.