Iterators

- References:
  - Text book : Chapter 15
  - Oracle/Sun Java Tutorial : http://download.oracle.com/javase/6/docs/api/index.html
  - Only cover usage of iterator in Java APIs.

1. What is an iterator?

- A program component
  - Enables you to step through, traverse a collection of data
  - Can tell whether next entry exists

- Iterator may be manipulated
  - Asked to advance to next entry
  - Give a reference to current entry
  - Modify the list as you traverse it

- This chapter covers **Java class library interfaces: Iterable, Iterator and ListIterator**

2. Java Class Library: the interface java.lang.Iterable

  - Refer to collection diagram: All classes in collections implement iterable interface

  - only define one method which returns an iterator over a set of elements of type T:

    Iterator<T> iterator()

    i.e. all classes need to implement Iterable class

- A class that implements the interface Iterable (and Iterator) also support special for-each loop statement. It is uses to traverse objects in an instance of the class

  Example:

```
ListWithIteratorInterface<String> nameList =
            new LinkedListWithIterator<String>();
nameList.add("Joe");
nameList.add("Jess");
nameList.add("Josh");
nameList.add("Jen");

for (String name : nameList)  // collection  classes implement this!
    System.out.print(name + " ");
System.out.println();
```

  Output:

```
    Joe Jess Josh Jen
```

3. Java Class Library: the interface java.util.Iterator

- Specifies three methods: hasNext(), next(), remove()
- Specifies a generic type for entries

```
package java.util;
public interface Iterator<T>
{

 /** Task: Detects whether the iterator has completed its traversal
  *       and gone beyond the last entry in the collection of data.
  *  @return true if the iterator has another entry to return */
 public boolean hasNext();
```

```
/** Task: Retrieves the next entry in the collection and
 *       advances the iterator by one position.
 *  @return a reference to the next entry in the iteration,
 *          if one exists
 *  @throws NoSuchElementException if the iterator had reached the
 *          end already, that is, if hasNext() is false */
public T next();

/** Task: Removes from the collection of data the last entry that
 *       next() returned. A subsequent call to next() will behave
 *       as it would have before the removal.
 *  Precondition: next() has been called, and remove() has not been
 *       called since then. The collection has not been altered
 *       during the iteration except by calls to this method.
 *  @throws IllegalStateException if next() has not been called, or
 *          if remove() was called already after the last call to
 *          next().
 *  @throws UnsupportedOperationException if this iterator does
 *          not permit a remove operation. */
public void remove(); // Optional method
} // end Iterator
```

- Position of an iterator is not at an entry
  - Positioned either before first entry
  - Or between two entries

- Examples: Usage of class iterator

  - Consider a list of names created with the code below

```
ListInterface<String> nameList = new LList<String>();
nameList.add("Jamie");
nameList.add("Joey");
nameList.add("Rachel");
```

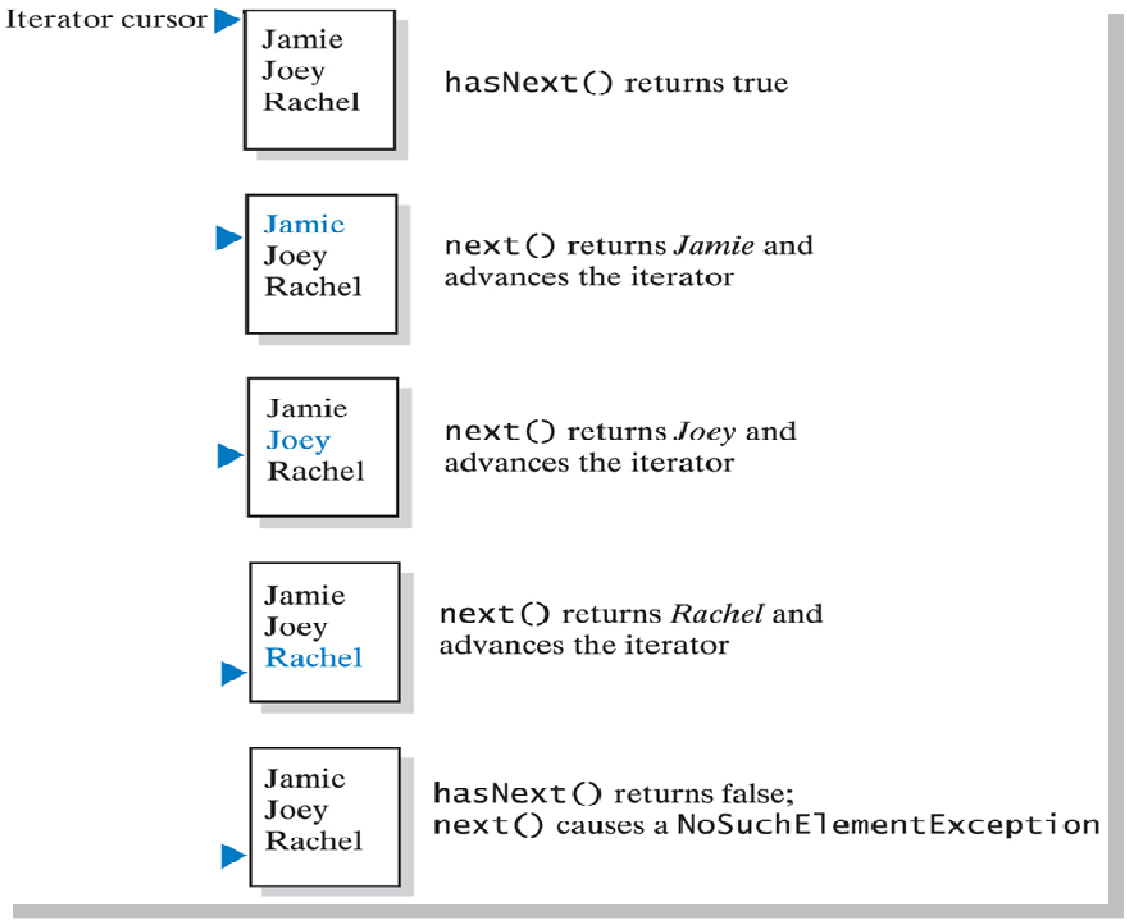
- Suppose that Iterator class is implemented
  We can use it as follows:

  Iterator<String> nameIterator = nameList.iterator();

  - returns an Iterator object "nameIterator" which refers to nameList object
  - position is "before" first entry

- Let look at some operations

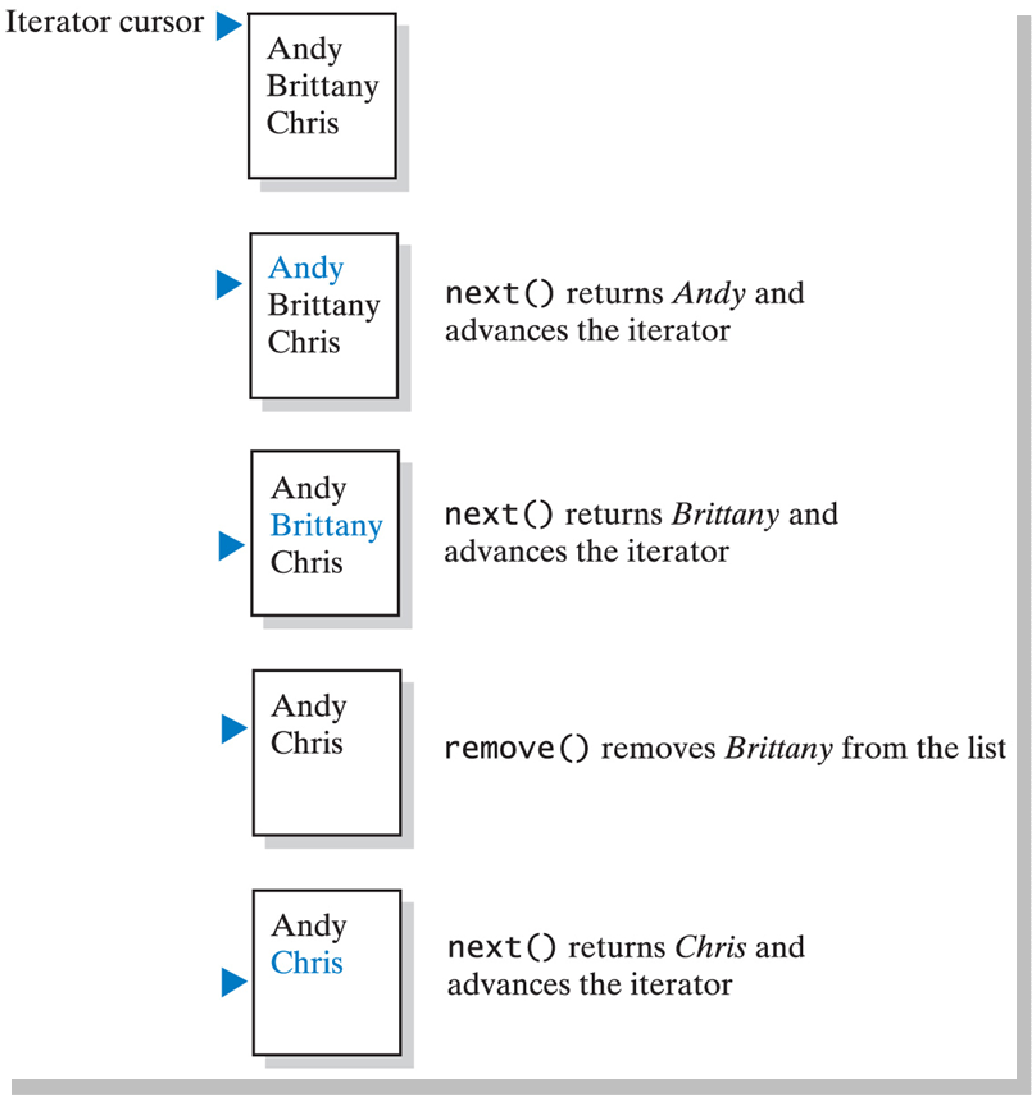
  nameIterator.hasNext() // returns true
  nameIterator.next() // returns the string Jamie and advances the iterator
  nameIterator.next() // returns the string Joey and advances the iterator
  nameIterator.next() // returns the string Rachel and advances the iterator
  nameIterator.hasNext() // returns false
  nameIterator.next() // causes a NoSuchElementException



Another example:

Assume the list contains names Andy, Brittany, Chris
With operations:

```
nameIterator.next();
nameIterator.next();
nameIterator.remove();
nameIterator.next();
```



4. Java Class Library: the interface java.util.ListIterator

- An alternative interface for iterators in Java Class Library

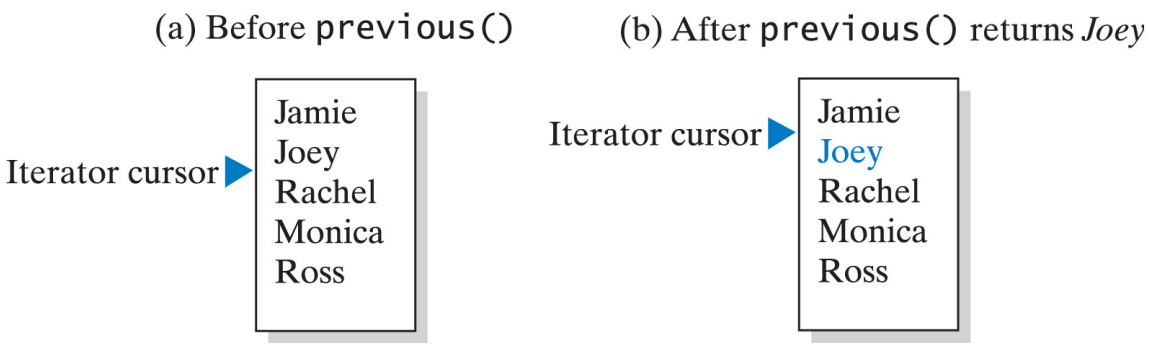
  public interface ListIterator<E> extends Iterator<E>

- Enables traversal in either direction

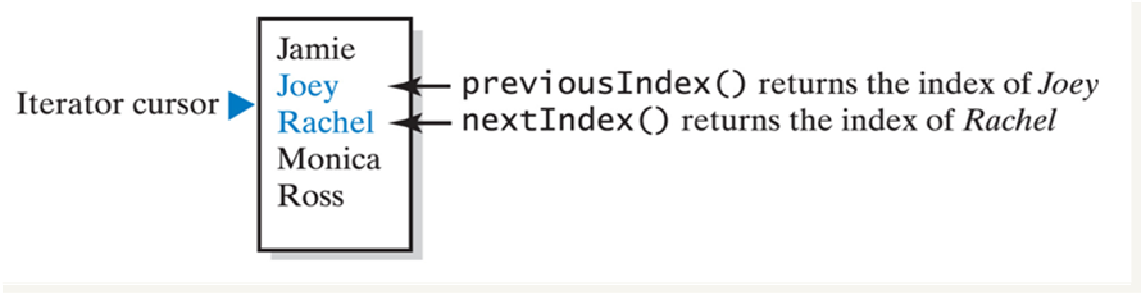
- Allows modification of list during iteration

```
package java.util;
public interface ListIterator<T> extends Iterator<T>
{
  public boolean hasNext();
  public T next();

  /** Task: Removes from the list the last entry that either next()
   *        or previous() has returned.
   *  Precondition: next() or previous() has been called, but the
   *        iterator remove() or add() method has not been called
   *        since then. That is, you can call remove only once per
   *        call to next() or previous(). The list has not been altered
   *        during the iteration except by calls to the iterator
   *        remove(), add(), or set() methods.
   *  @throws IllegalStateException if next() or previous() has not
   *          been called, or if remove() or add() has been called
   *          already after the last call to next() or previous()
   *  @throws UnsupportedOperationException if this iterator does not
   *          permit a remove operation */
  public void remove(); // Optional method

// The previous three methods are in the interface Iterator; they are
// duplicated here for reference and to show new behavior for remove.

//more in next few slides…
```

- New methods
  - hasPrevious // vs hasNext()
  - previous // vs next()
  - nextIndex // position of subsequent next() entry, starting with 0
  - previoudIndex // position of subsequent previous() entry
  - add
  - set




```
/** Task: Detects whether the iterator has gone before the first
 *       entry in the list.
 *  @return true if the iterator has another entry to visit when
 *          traversing the list backward; otherwise returns false */
public boolean hasPrevious();

/** Task: Retrieves the previous entry in the list and moves the
 *       iterator back by one position.
 *  @return a reference to the previous entry in the iteration, if
 *          one exists
 *  @throws NoSuchElementException if the iterator has no previous
 *          entry, that is, if hasPrevious() is false */
public T previous();
```




```
/** Task: Gets the index of the next entry.
 *  @return the index of the list entry that a subsequent call to
 *          next() would return. If next() would not return an entry
 *          because the iterator is at the end of the list, returns
 *          the size of the list. Note that the iterator numbers
 *          the list entries from 0 instead of 1. */
public int nextIndex();

/** Task: Gets the index of the previous entry.
 *  @return the index of the list entry that a subsequent call to
 *          previous() would return. If previous() would not return
 *          an entry because the iterator is at the beginning of the
 *          list, returns -1. Note that the iterator numbers the
 *          list entries from 0 instead of 1. */
public int previousIndex();
```

Add("Ashley")

Before		After Add
Jen		Jen
→		Ashley
Jim	→	
Josh		Jim
		Josh

```
 /** Task: Adds an entry to the list just before the entry, if any,
  *       that next() would have returned before the addition. This
  *       addition is just after the entry, if any, that previous()
  *       would have returned. After the addition, a call to
  *       previous() will return the new entry, but a call to next()
  *       will behave as it would have before the addition.
  *       Further, the addition increases by 1 the values that
  *       nextIndex() and previousIndex() will return.
  *  @param newEntry  an object to be added to the list
  *  @throws ClassCastException if the class of newEntry prevents the
  *          addition to this list
  *  @throws IllegalArgumentException if some other aspect of newEntry
  *          prevents the addition to this list
  *  @throws UnsupportedOperationException if this iterator does not
  *          permit an add operation */
 public void add(T newEntry); // Optional method


 /** Task: Replaces the last entry in the list that either next()
  *       or previous() has returned.
  *  Precondition: next() or previous() has been called, but the
  *       iterator  remove() or add() method has not been called
  *       since then.
  *  @param newEntry  an object that is the replacement entry
  *  @throws ClassCastException if the class of newEntry prevents the
  *          addition to this list
  *  @throws IllegalArgumentException if some other aspect of newEntry
  *          prevents the addition to this list
  *  @throws IllegalStateException if next() or previous() has not
  *          been called, or if remove() or add() has been called
  *          already after the last call to next() or previous()
  *  @throws UnsupportedOperationException if this iterator does not
  *          permit a set operation */
 public void set(T newEntry); // Optional method
} // end ListIterator
```

- Recall interface java.util.List

  It extends Iterable: Thus has method iterator

  Also have methods:
  listIterator() :
  Returns a list iterator of the elements in this list (in proper sequence).
  listIterator(int index) :
  Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.

- Example: Assume nameList contains : Jess, Jim, Josh

```
ListIterator<String>  traverse = nameList.listIterator(0);

System.out.println("nextIndex     " + traverse.nextIndex());
System.out.println("hasNext       " + traverse.hasNext());
System.out.println("previousIndex " + traverse.previousIndex());
System.out.println("hasPrevious   " + traverse.hasPrevious());
Output:         0, true, -1, false

System.out.println("next      " + traverse.next());
System.out.println("nextIndex " + traverse.nextIndex());
System.out.println("hasNext   " + traverse.hasNext());
Ouput:          Jess, 1, true

System.out.println("previousIndex " + traverse.previousIndex());
System.out.println("hasPrevious   " + traverse.hasPrevious());
System.out.println("previous      " + traverse.previous());
System.out.println("nextIndex     " + traverse.nextIndex());
System.out.println("hasNext       " + traverse.hasNext());
System.out.println("next          " + traverse.next());
Output:         0, true, Jess, 0, true, Jess

traverse.set("Jen");  // last operation was next()

List become:    Jen, Jim, Josh
```