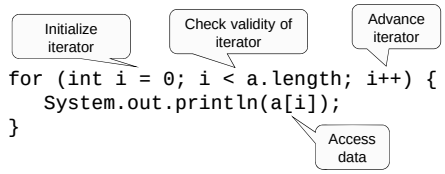
CSIS 3103

Ch 2: Iterators

The Iterator Pattern

An iterator is like a moving place marker that keeps track of the current position in a list



Iterator pattern for array-like structures

The Iterator<E> Interface

The `Iterator` interface specifies objects that can iterate through the elements of a collection

Method	Behavior
`boolean hasNext()`	Returns `true` if the `next` method returns a value.
`E next()`	Returns the next element. If there are no more elements, throws the `NoSuchElementException`.
`void remove()`	Removes the last element returned by the `next` method.

Combines the *access data* and *advance iterator* steps

Which is more efficient?

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

or

```
for (int i = 0; i < list.size(); i++) {
   int value = list.get(i);
   // Do something with value
}
```

The `Iterable` Interface

- Requires an implementing class to provide an `iterator` method
- The `Collection` interface extends the `Iterable` interface

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

Factory method: A method that creates a new object whose actual type is unknown

Implementing an Iterator

Usually done with in a nested inner class

```
private static class IterImpl<E>
                        implements Iterator<E> {

}

public Iterator<E> iterator() {
   return new IterImpl<E>(head);
}
```

The Enhanced ***for*** Statement

`for` (*formalParam : collection) { … }*

```
for (String nextStr : myList) { … }
```

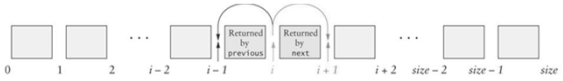
During each loop iteration, *formalParam* references the next element of *collection*, starting with the first element. The *collection* must be an array or a class that implements the `Iterable` interface (includes all classes that implement `Collection`).

`Iterator` limitations

- Can only traverse a `List` in the forward direction
- Provides a `remove` method but no `add`
- Must start at the beginning of the list

  `ListIterator<E>` is an extension of `Iterator<E>` that overcomes these limitations

  `ListIterator` positions are assigned an index from 0 to `size`



`ListIterator<E>` Interface

Method	Behavior
`void add(E obj)`	Inserts object `obj` into the list just before the item that would be returned by the next call to method `next` and after the item that would have been returned by method `previous`. If method `previous` is called after `add`, the newly inserted object will be returned.
`boolean hasNext()`	Returns `true` if `next` will not throw an exception.
`boolean hasPrevious()`	Returns `true` if `previous` will not throw an exception.
`E next()`	Returns the next object and moves the iterator forward. If the iterator is at the end, the `NoSuchElementException` is thrown.
`int nextIndex()`	Returns the index of the item that will be returned by the next call to `next`. If the iterator is at the end, the list size is returned.
`E previous()`	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the `NoSuchElementExcepton` is thrown.
`int previousIndex()`	Returns the index of the item that will be returned by the next call to `previous`. If the iterator is at the beginning of the list, −1 is returned.
`void remove()`	Removes the last item returned from a call to `next` or `previous`. If a call to `remove` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.
`void set(E obj)`	Replaces the last item returned from a call to `next` or `previous` with `obj`. If a call to `set` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.

ListIterator<E> Interface

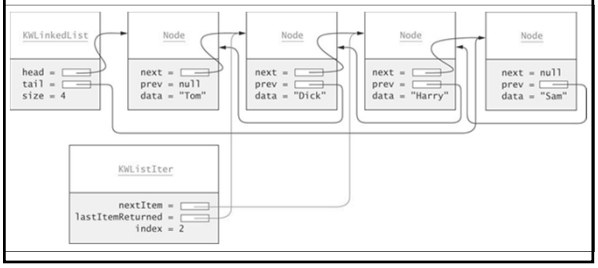
Method	Behavior
`public ListIterator<E> listIterator()`	Returns a `ListIterator` that begins just before the first list element.
`public ListIterator<E> listIterator(int index)`	Returns a `ListIterator` that begins just before position `index`.

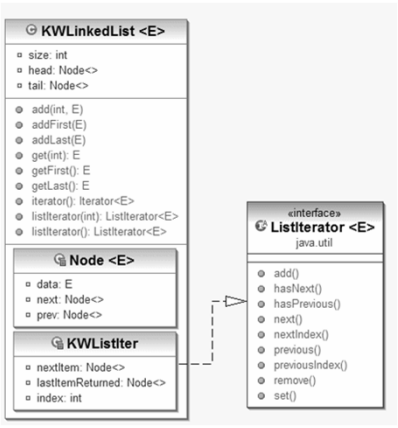
`Iterator` and `ListIterator`

- `Iterator` interface requires fewer methods and can be used to iterate over more general data structures but only in one direction
- `Iterator` is required by `Collection` classes
- `ListIterator` is required only by the `List` interface

`KWLinkedList`

A class that partially implements the `List` interface using a double-linked list with a `ListIterator`





`AbstractCollection`, `AbstractList`, `AbstractSequentialList`

- The Java API includes these "helper" abstract classes to help build implementations of their corresponding interfaces
- Helper classes provide implementations for interface unused methods so programmer can extend the abstract classes and implement only the desired methods

Implementing a Subclass of `Collection<E>`

Extend `AbstractCollection<E>` by implementing only:

- `add(E)`
- `size()`
- `iterator()`
- An inner class that implements `Iterator<E>`

Implementing a Subclass of `List<E>`

Extend `AbstractList<E>` by implementing only:

- `add(int, E)`
- `get(int)`
- `remove(int)`
- `set(int E)`
- `size()`

- `AbstractList` implements `Iterator<E>`

`List` and `RandomAccess` Interfaces

- Accessing a `LinkedList` using an index requires an $O(n)$ traversal of the list until the index is located
- The `RandomAccess` interface is applied to list implementations in which indexed operations are efficient (e.g. `ArrayList`)