Iterators

Motivation

- We often want to access every item in a data structure or collection in turn
  - We call this ***traversing*** or ***iterating over*** or ***stepping through*** or ***visiting every item in*** the data structure or collection
- Example with a data structure (array):

```
for (int i = 0; i < arr.length(); i++)
    /*  do something to arr[i]   */
```

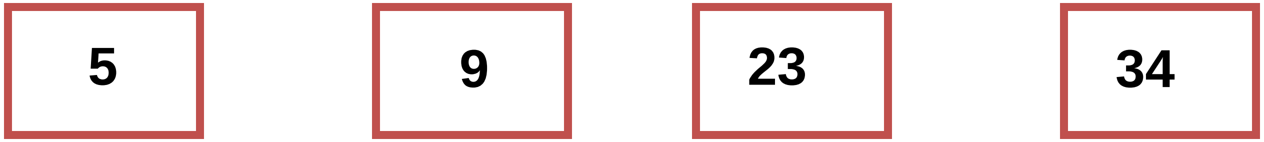
  - This is straighforward because we know exactly how an array works!

Motivation

- What if we want to traverse a *collection* of objects?
  - A list, a stack, a queue …
  - Its underlying implementation may not be known to us
- Java provides a *common scheme* for stepping through all elements in *any* collection, called an ***iterator***

What is an Iterator?

- An ***iterator*** is a mechanism used to step through the elements of a collection one by one
  - Each element is "***delivered*** " exactly once
- ***Example***
  - Iterate through an ordered list and print each element in turn



5

9

23

34

Iterator Interface

- The Java API has a generic interface called Iterator<T> that specifies what methods are required of an iterator
  - **public boolean hasNext( );**
    returns true if there are more elements in the iteration
  - **public T next( );**
    returns the next element in the iteration
  - **public void remove( );**
    removes the last element returned by the iterator (*optional operation*)

Array Iterator

- If we had a collection with an array implementation, we would need an ***array implementation*** of the **Iterator** interface
    - Its attributes
    - Its constructor
    - The code for the methods hasNext and next
        - In what order does it deliver the items?

- ***Note:*** ***This code*** *can be used by an array implementation of* ***any*** *collection!*

```
// Represents an iterator over the elements of an array

import java.util.*;

public class ArrayIterator<T> implements Iterator<T> {

    // Attributes
    private int count;   // number of elements in collection
    private int current; // current position in the iteration
    private T[ ] items;   // items in the collection

    // Constructor: sets up this iterator using the
    // specified items
    public ArrayIterator (T[ ] collection, int size) {
            items = collection;
            count = size;
    current = 0;
    }
   // cont'd..
   // cont’d..
    // Returns true if this iterator has at least one
    // more element to deliver in the iteration
    public boolean hasNext( )  {
          return (current < count);
    }

    // Returns the next element in the iteration.
    // If there are no more elements in this iteration,
    // throws an exception.
    public T next( )  {
       if (! hasNext( ))
          throw new NoSuchElementException( );
     current++;
     return items[current - 1];
     }
}
```

Linked Iterator

- If we had a collection with a linked implementation, we would need a ***linked implementation*** of the **Iterator** interface
  - Its attributes
  - Its constructor
  - The code for the methods hasNext and next
    - In what order does it deliver the items?

- ***Note:** Again the code can be used by a linked implementation of **any** collection!*

```
import java.util.*;
public class LinkedIterator<T> implements Iterator<T> {

  // Attributes
   private LinearNode<T> current; // current position

   //  Constructor: Sets up this iterator using the specified items
   public LinkedIterator (LinearNode<T> collection){
            current = collection;
    }   //cont'd..
  // ..cont'd..
  //  Returns true if this iterator has at least one more element
  //  to deliver in the iteration.
  public boolean hasNext( ) {
    return (current!= null);
  }
  //  Returns the next element in the iteration. If there are no
  //  more elements in this iteration, throws an exception.
  public T next( ) {
    if (! hasNext( ))
      throw new NoSuchElementException( );
    T result = current.getElement( ); // ummm redesign?
    current = current.getNext( );
    return result;
  }
}
```

Iterators for a Collection

The last piece ….

- ***operation*** called iterator( )

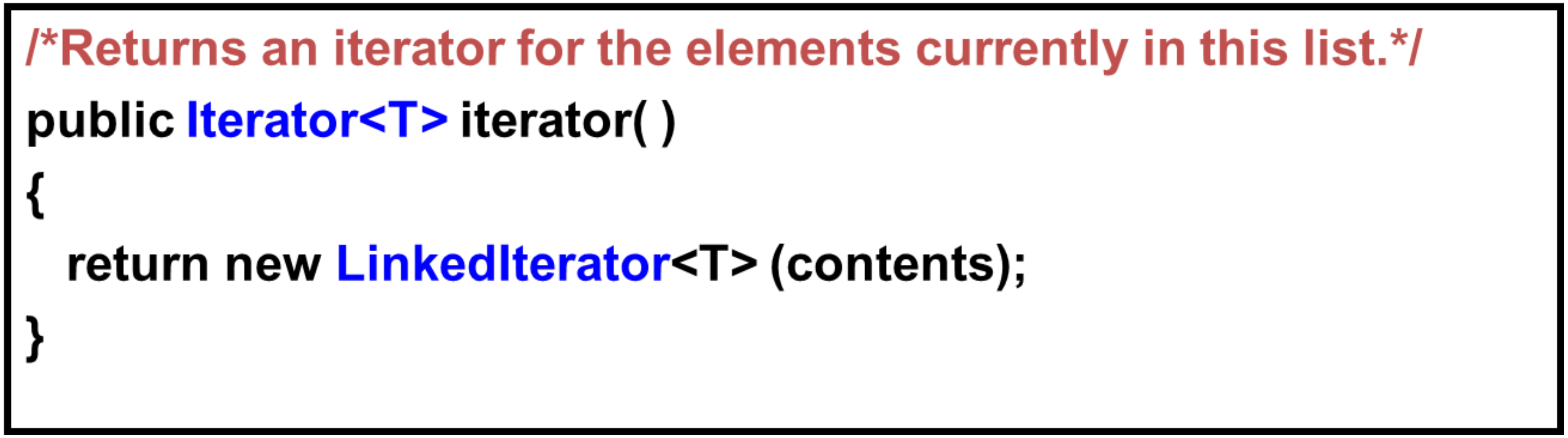
```
// Returns an iterator for the elements in this list
   public Iterator<T> iterator( );
```

The iterator Operation

- Note that the return type of the iterator operation is Iterator<T>
  - But Iterator<T> is an interface, not a class!
  - When the return type of a method is an *interface name*, the method actually returns an object from *a class that implements the interface*
    - The iterator operation in ArrayList will use the class ArrayIterator
    - The iterator operation in LinkedList will use the class LinkedIterator

iterator methods

```
/** Returns an iterator for the elements currently in this array.
*/
   public Iterator<T> iterator()
   {
      return new ArrayIterator<T> (list, rear);
   }
```



```
/*Returns an iterator for the elements currently in this list.*/
public Iterator<T> iterator( )
{
   return new LinkedIterator<T> (contents);
}
```

Using an Iterator

- When the iterator() method in a collection is invoked, it returns an “iterator object”

- We can then invoke the methods hasNext() and next() on that object, to iterate through the collection

Using an Iterator in an Application

```
  AUList<Person> myList = new AUList<Person>();

// Use iterator to display contents of list

  Iterator<Person> iter = myList.iterator();

  while(iter.hasNext() )
  {
      System.out.println(iter.next());
  }
```

Example: Using an Iterator within a Class Definition

- Rewrite the toString() method of ArrayList using its iterator:

Remember this means the current object

```
public String toString() {
    String result = “”;

    Iterator<T> iter  =  this.iterator();

    while ( iter.hasNext() )
       result = result  +  iter.next().toString() +  “\n”;

    return result;
}
```