Advanced Programming Techniques in Java

COSI 12B

List Interface & Linked Lists

Lecture 19

Class Objectives

- `List` (first subsection 11.1)
- Linked List (second subsection of 11.1)
- Iterators (third subsection of 11.1)

Review: Algorithm growth rates

- We measure runtime in proportion to the input data size, n
  - **Growth rate**: Change in runtime as n changes
- Say an algorithm runs $\mathbf{0.4n^3 + 25n^2 + 8n + 17}$
  - Consider the runtime when n is extremely large
  - We ignore constants like 25 because they are tiny next to n
  - The highest-order term (n^3) dominates the overall runtime
  - We say that this algorithm runs ”in the order of" n^3
  - or $\mathbf{O(n^3)}$ for short ("**Big-Oh** of n^3")

- **Big-Oh It's a measure of the longest amount of time it could possibly take for the algorithm to complete (upper bound)**

Review: Complexity classes

- **Complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size n

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Review: Collection efficiency

- Efficiency of various operations on `ArrayList`:

Method	**ArrayList**
`add`	O(1)
`add(`**`index`**`, `**`value`**`)`	O(n)
`indexOf`	O(n)
`get`	O(1)
`remove`	O(n)
`set`	O(1)
`size`	O(1)

Review: Comparable implementation

```
// The CalendarDate class stores information about a single calendar
   date (month and day but no year).

public class CalendarDate implements Comparable<CalendarDate> {
    private int month;
    private int day;

    public CalendarDate(int month, int day) {
        this.month = month;
        this.day = day;
    }
  public int compareTo(CalendarDate other) {
        if (this.month != other.month) {
            return this.month - other.month;
        } else {
            return this.day - other.day;
        }
    }
    public String toString() {
        return this.month + "/" + this.day;
    }
}
```

Compares this calendar date to another date.Dates are compared by month and then by day.

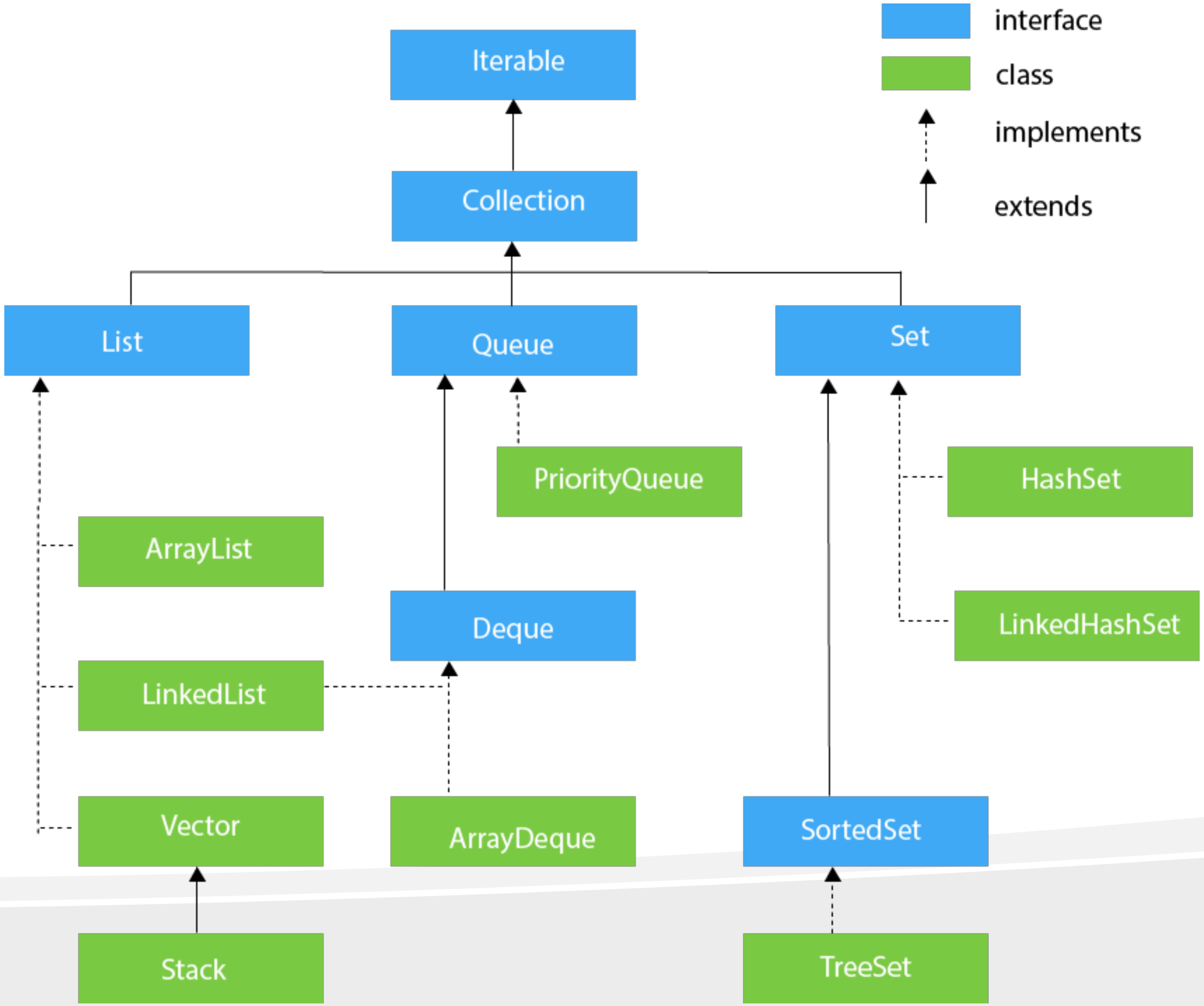
Review: Example Client Program

```
// Short program that creates a list of the birthdays of the first 5
// US Presidents and that puts them into sorted order.

import java.util.*;

public class CalendarDateTest {
    public static void main(String[] args) {
        ArrayList<CalendarDate> dates = new ArrayList<CalendarDate>();
        dates.add(new CalendarDate(2, 22));
        dates.add(new CalendarDate(10, 30));
        dates.add(new CalendarDate(4, 13));
        dates.add(new CalendarDate(3, 16));
        dates.add(new CalendarDate(4, 28));

        System.out.println("birthdays before sorting = " + dates);
        Collections.sort(dates);
        System.out.println("birthdays after sorting = " + dates);
    }
}
```

since CalendarDate implements the Comparable we can use the Collections.sort method

OUTPUT:
birthdays before sorting = [2/22, 10/30, 4/13, 3/16, 4/28]
birthdays after sorting = [2/22, 3/16, 4/13, 4/28, 10/30]

Review: Collections Framework Diagram

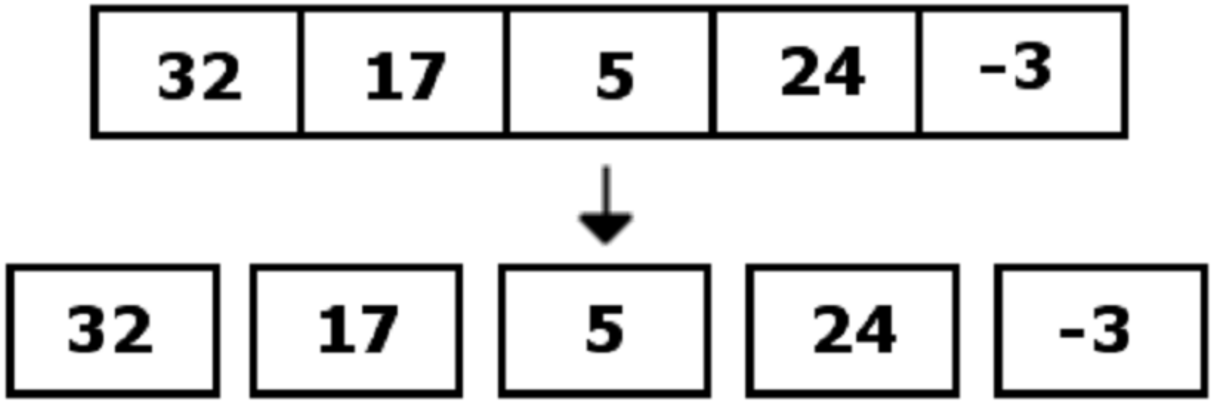


Internals of `ArrayList`

- It is internally stored in an array. So, it has a certain capacity
  - Capacity: size of the array used to store the elements in the list
- It is always at least as large as the list size
- As elements are added to an `ArrayList`, its capacity grows “automatically”
- Random access to elements
  - `set/get` an element to/at specific index position has constant time, **O(1)**
  - To `add` at the end of the `ArrayList` it takes **O(1)**
- Remove, or add at a specified index operations have linear time **O(n)**

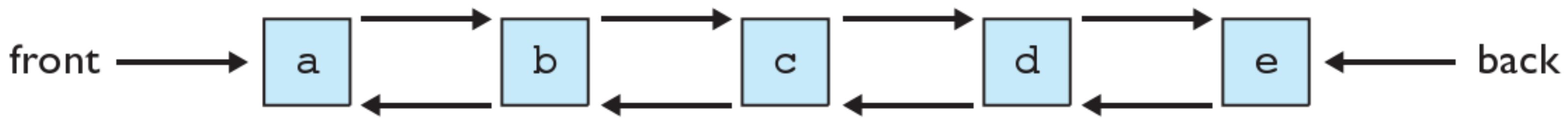
The underlying issue

- The elements of an `ArrayList` are too tightly attached; can't easily rearrange them
- Can we break the element storage apart into a more dynamic and flexible structure?



Linked list

- **Linked list** is a list implemented using a linked sequence of values
  - Each value is stored in a small object called a **node**, which also contains references to its neighbor nodes
  - The list keeps a reference to the first and/or last node
  - In Java, represented by the class LinkedList



`LinkedList` usage example

- A `LinkedList` can be used much like an `ArrayList`
  - It also implements the `List` interface, so it offers the methods of that interface

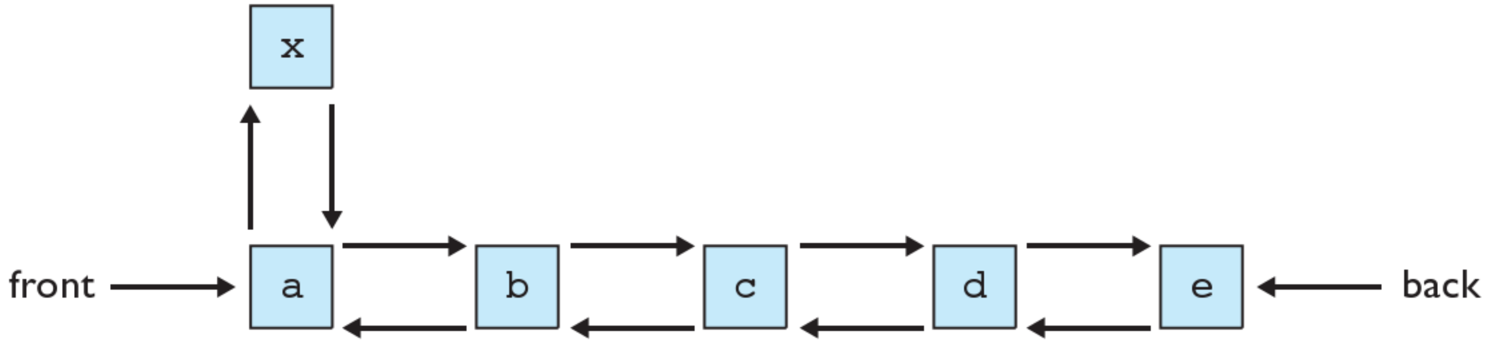
```
LinkedList <String> words = new LinkedList<String>();
words.add("hello");
words.add("goodbye");
words.add("this");
words.add("that");
```

- Advantage: elements are added/removed at/from the front and back of the list quickly
  - There’s no shifting, we just create a new node object and links it with the others

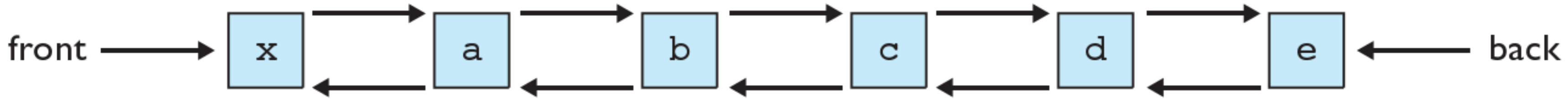
Adding elements to the list

1. Make a new node to hold the new element.

2. Connect the new node to the other nodes in the list.

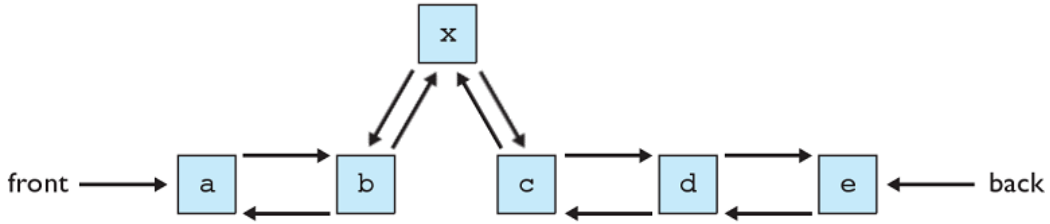


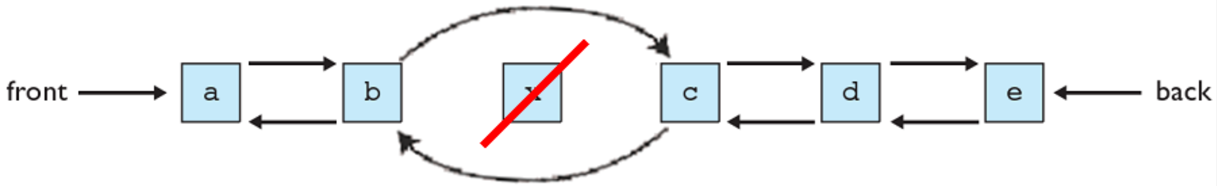
3. Change the front of the list to point to the new node.



Linked list performance

- To add, remove, get a value at a given index:
  - The list must advance through the list to the node just before the one with the proper index
  - For example, to add a new value to the list, the list creates a new node, walks along its existing node links to the proper index, and attaches it to the nodes that should precede and follow it
  - This is very fast when adding to the front or back of the list (because the list contains references to these places), but slow elsewhere





Linked List Implementation – Inner Classes

```
public class myLinkedList<E> {
  private Node<E> head;
  private Node<E> tail;
  private int size;

  private static class Node<E>{
          private E data;
          private Node<E> next;
          private Node<E> previous;

          private Node(E dataItem)  {
              data = dataItem;
              next = null;
              previous = null;
            }
  }
}
```

Generally, all details of the `Node` class should be private. This applies also to the data fields and constructors.

The keyword `static` indicates that the `Node<E>` class will not reference its outer class

Static inner classes are also called *nested classes*

A particularly slow idiom

```
List<String> list = new LinkedList<String>();
// ... (put a lot of data into the list)

// print every element of linked list
for (int i = 0; i < list.size(); i++) {
    String element = list.get(i);
    System.out.println(i + ": " + element);
}
```

- This code executes a slow operation (`get`) every pass through a loop that runs many times
  - This code will take long time to run for large data sizes
  - **Sequential access is** slower than the **random access** of `ArrayLists`

The problem of position

- The code on the previous slide is wasteful because it throws away the position each time
  - Every call to `get` has to re-traverse the list
- It would be much better if we could somehow keep the place in the list at each index as we looped through it
- Java uses special objects to represent a position of a collection as it's being examined
  - These objects are called **iterators**

`Collection` Interface

- Defines fundamental methods
  - `Iterator iterator();`
  - …
- Provides an `Iterator` to step through the elements in the `Collection`

`Iterator<E>` Interface

- Defines three fundamental methods
  - `E next()` returns the next element. If there are no more elements, throws `NoSuchElementExecption`
  - `boolean hasNext()` returns `true` if there is another element to process
  - `void remove()` removes the last element returned by the `next` method (must be preceded by a call to `next`)
- These three methods provide access to the contents of the collection
- An `Iterator` knows position within a collection
- Each call to `next()` "reads" an element from the collection
  - Then you can use it or remove it

Iterator

- The easiest way to cycle through the elements in a collection is to employ an iterator

```
Iterator<type> iter = collection.iterator();
while (iter.hasNext()){
          type nextElement = iter.next();
          // do something with nextElement
}
```

Iterator

```
import java.util.* ;
public class IteratorExample {
     public static void main ( String[] args) {
          // Create and populate the list
          ArrayList<String> names = new ArrayList<String>();
          names.add( "Jan" ); names.add( "Levi" );
          names.add( "Tom" ); names.add( "Jose" );

          // Create an iterator for the list
          Iterator<String> iter = names.iterator();

          // Use the iterator to visit each element
          while ( iter.hasNext() )
                    System.out.println( iter.next() );
     }
}
```

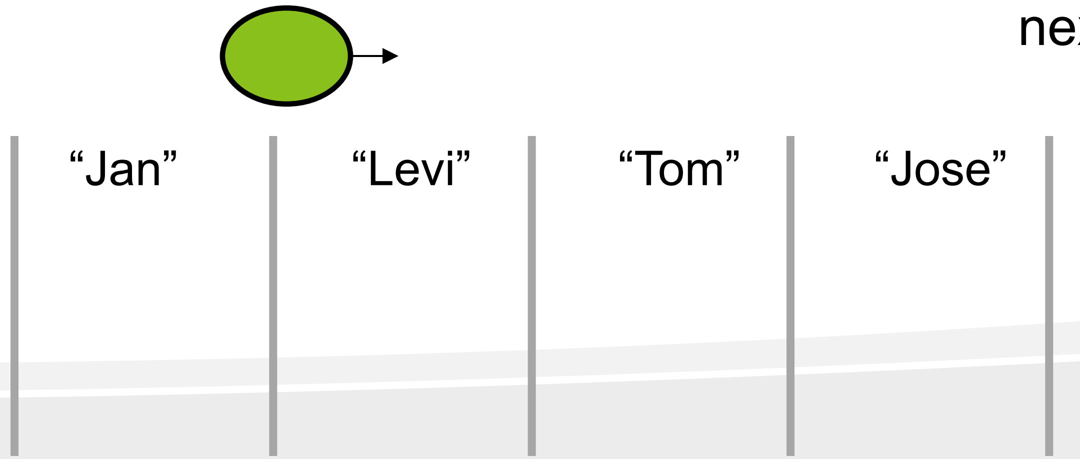
Iterator Position

```
ArrayList<String> names = new ArrayList<String>();
names.add("Jan");
names.add("Levi");
names.add("Tom");
names.add("Jose");
Iterator<String> it = names.iterator();
int i = 0;
```

Iterator Position

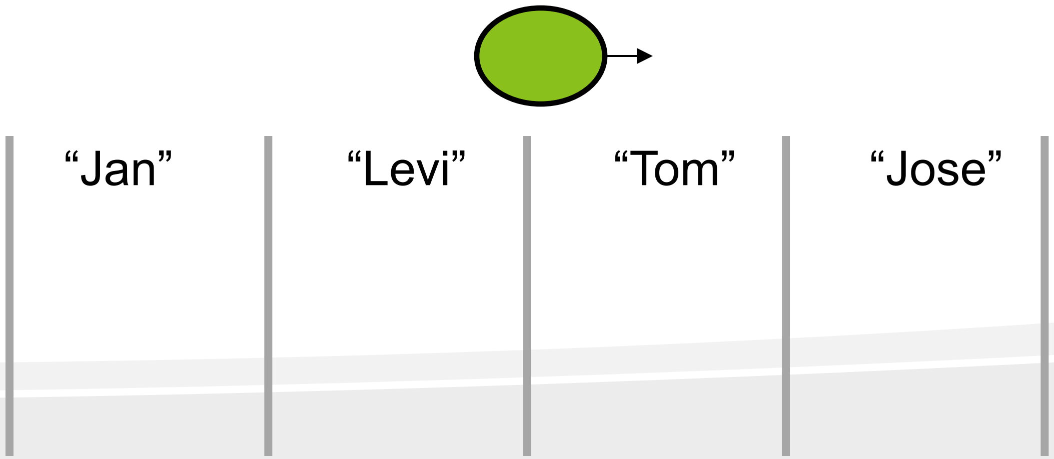
```
while( it.hasNext() ) {
    i++;
    System.out.println( it.next() );
}
// when i == 1, prints out Jan
```

first call to `next` moves iterator to next post and returns “Jan”



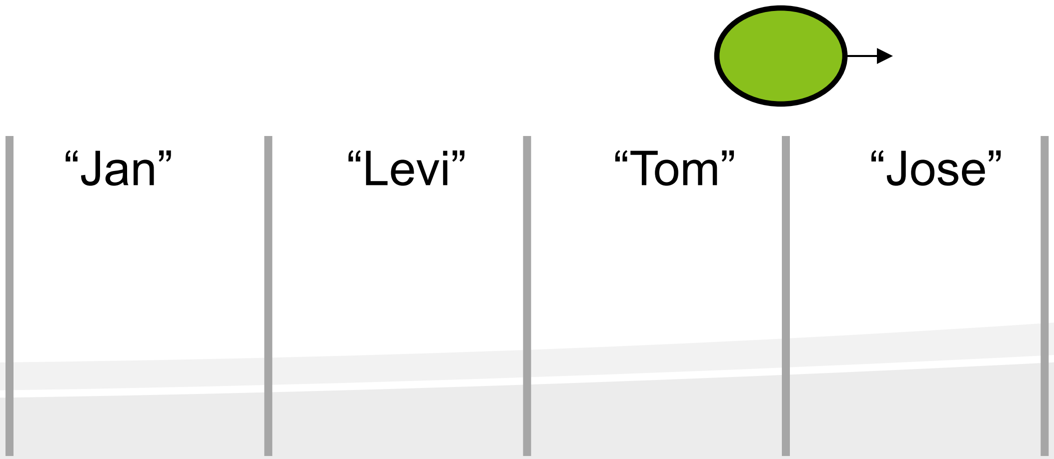
Iterator Position

```
while( it.hasNext() ) {
    i++;
    System.out.println( it.next() );
}
// when i == 2, prints out Levi
```



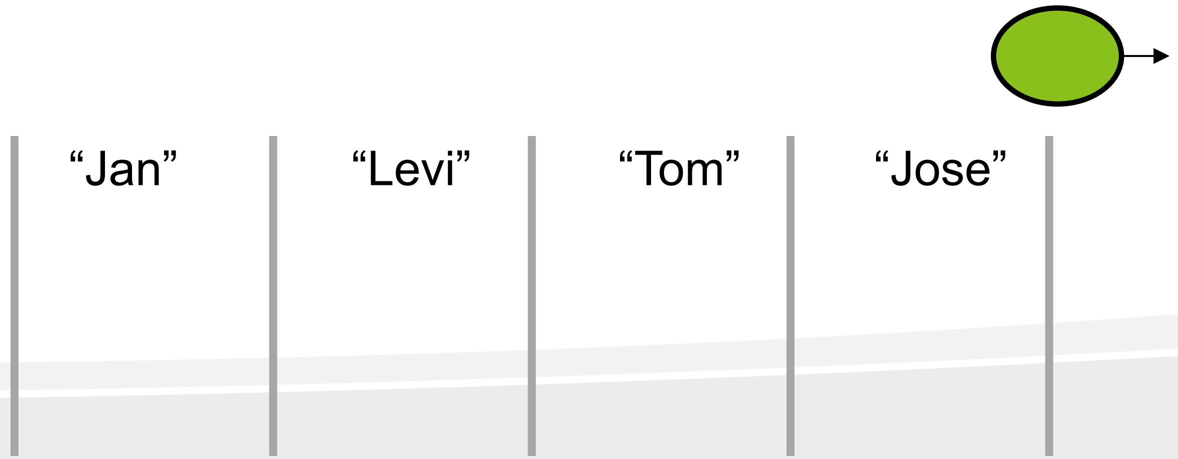
Iterator Position

```
while( it.hasNext() ) {
     i++;
     System.out.println( it.next() );
}
// when i == 3, prints out Tom
```



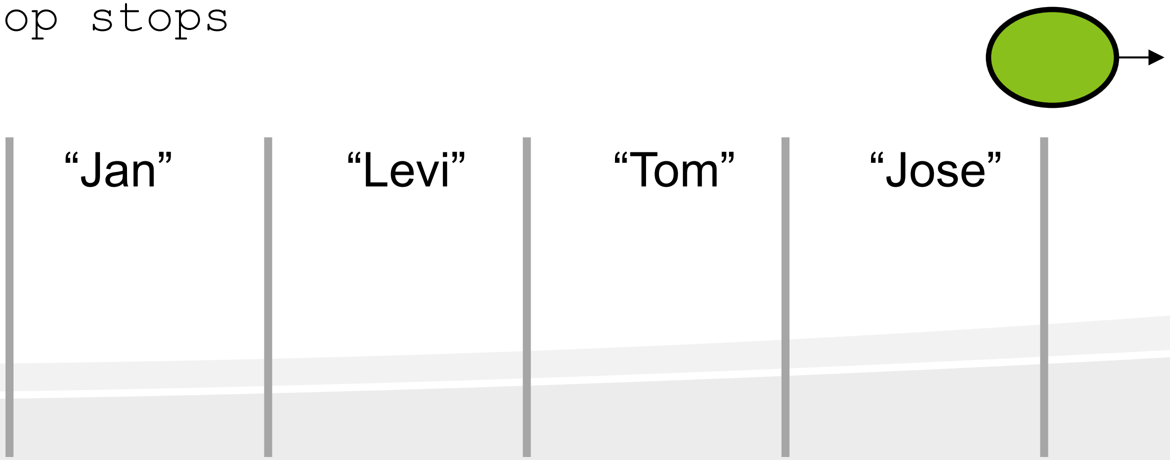
Iterator Position

```
while( it.hasNext() ) {

    i++;

    System.out.println( it.next() );

}

// when i == 4, prints out Jose
```



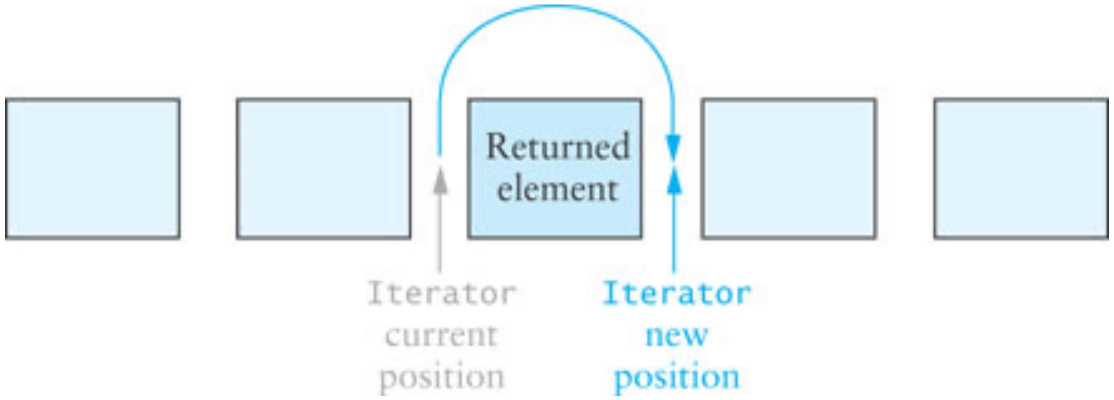
Iterator Position

```
while( it.hasNext() ) {
     i++;
     System.out.println( it.next() );
}
// call to hasNext returns false
// while loop stops
```



Iterator Position

- An `Iterator` is conceptually *between* elements; it does not refer to a particular object at any given time



Fixing the slow LL idiom

```
// print every element of the list
for (int i = 0; i < list.size(); i++) {
    Object element = list.get(i);
    System.out.println(i + ": " + element);
}
```

```
// print every element of the list
Iterator<String> itr = list.iterator();
for (int i = 0; itr.hasNext(); i++) {
    String element = itr.next();
    System.out.println(i + ": " + element);
}
```

- What’s the big-O now?

Another “slow” example

```
//input: LinkedList<String> list
int i=0;
while (i<list.size()){
  String element = list.get(i);
  if (element.length()%2 == 0)  {
        list.remove(i);
  else {
        i++; // skip to next element
  }
}
```

slow performance for large files

slow performance for large files

Iterator usage

- Iterator template syntax:

```
Iterator<E> itr = <collection>.iterator();
while (itr.hasNext()) {
    <do something with itr.next() >;
}
```

- Remove all strings with an even number of characters from a linked list:

```
// removes all strings of even length from the list
public static void removeEvenLength(LinkedList<String> list) {
    Iterator<String> i = list.iterator();
    while (i.hasNext()) {
        String element = i.next();
        if (element.length() % 2 == 0) {
            i.remove();
        }
    }
}
```

Benefits of iterators

- Speed up loops over lists' elements
  - Implemented for both `ArrayLists` and `LinkedLists`
  - Makes more sense to use it for `LinkedLists` since `get` operations is cheap in `ArrayList`
- A unified way to examine all elements of a collection
  - Every collection in Java has an `iterator` method
  - In fact, that's the *only* guaranteed way to examine the elements of any `Collection`
- Don't have to use indexes

Iterator is still not perfect

```
// add a 0 after any odd element
Iterator<Integer> itr = list.iterator();
int i = 0;
while (itr.hasNext()) {
    int element = itr.next();
    if (element % 2 == 1) {
        list.add(i, 0));    // fails
    }
}
```

- We can't use the iterator to add or set elements
  - The iterator is programmed to crash if the list is modified externally while the iterator is examining it

Concurrent modification exception

```
public void doubleList(LinkedList<Integer> list) {
    Iterator<Integer> i = list.iterator();
    while (i.hasNext()) {
        int next = i.next();
        list.add(next);  // ConcurrentModificationException
    }
}
```

- While you are still iterating, you cannot call any methods on the list that modify the list's contents
  - The code crashes with a `ConcurrentModificationException`
  - It is okay to call a method on the **iterator itself** that modifies the list (`remove`)
  - `get` and `remove` loops ONLY (not `set`/`add` operations)

The `ListIterator<E>` interface

- Extends the `Iterator` interface

- The `LinkedList` class implements the `List<E>` interface using a doubly-linked list

- Methods in `LinkedList` that return a `ListIterator`:
  - `public ListIterator<E> listIterator()`
  - `public ListIterator<E> listIterator(int index)`

- Methods in the `ListIterator` interface:
  - `add, hasNext, hasPrevious, next, previous, nextIndex, previousIndex, remove, set`

`Collections` class (not the `Collection` interface)

- The following static methods in the `Collections` class operate on either type of list
  - Example: `Collections.replaceAll(list, "hello", "goodbye");`

Method name	Description
`binarySearch(`*list, value*`)`	searches a sorted list for a value and returns its index
`copy(`*dest, source*`)`	copies all elements from one list to another
`fill(`*list, value*`)`	replaces all values in the list with the given value
`max(`*list*`)`	returns largest value in the list
`min(`*list*`)`	returns smallest value in the list
`replaceAll(`*list, oldValue, newValue*`)`	replaces all occurrences of *oldValue* with *newValue*
`reverse(`*list*`)`	reverses the order of elements in the list
`rotate(`*list, distance*`)`	shifts every element's index by the given distance
`sort(`*list*`)`	places the list's elements into natural sorted order
`swap(`*list, index1, index2*`)`	switches element values at the given two indexes