Advanced Programming Techniques in Java

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 s
  - **Growth rate**: Change in runtime as n changes
- Say an algorithm runs **$0.4n^3 + 25n^2 + 8n + 17$**
  - Consider the runtime when n is extremely large
  - We ignore constants like 25 because they are tiny nex
  - The highest-order term (n^3) dominates the overall runt
  - We say that this algorithm runs ”in the order of" n^3
  - or **$O(n^3)$** for short ("**Big-Oh** of n^3")

- **Big-Oh It's a measure of the longest amount of ti**
  algorithm to complete (upper bound)

Review: Complexity classes

- **Complexity class**: A category of algorithm efficienc
  the input size n

Big-O	Name
$O(1)$	Consta
$O(\log n)$	Logarit
$O(n)$	Linear
$O(n \log n)$	Log-lin
$O(n^2)$	Quadra
$O(n^3)$	Cubic
$O(2^n)$	Expone
$O(n!)$	Factori

Review: Collection efficiency

- Efficiency of various operations on `ArrayList`:

Method
`add`
`add(`**`index, value`**`)`
`indexOf`
`get`
`remove`
`set`
`size`

Review: Comparable implementat

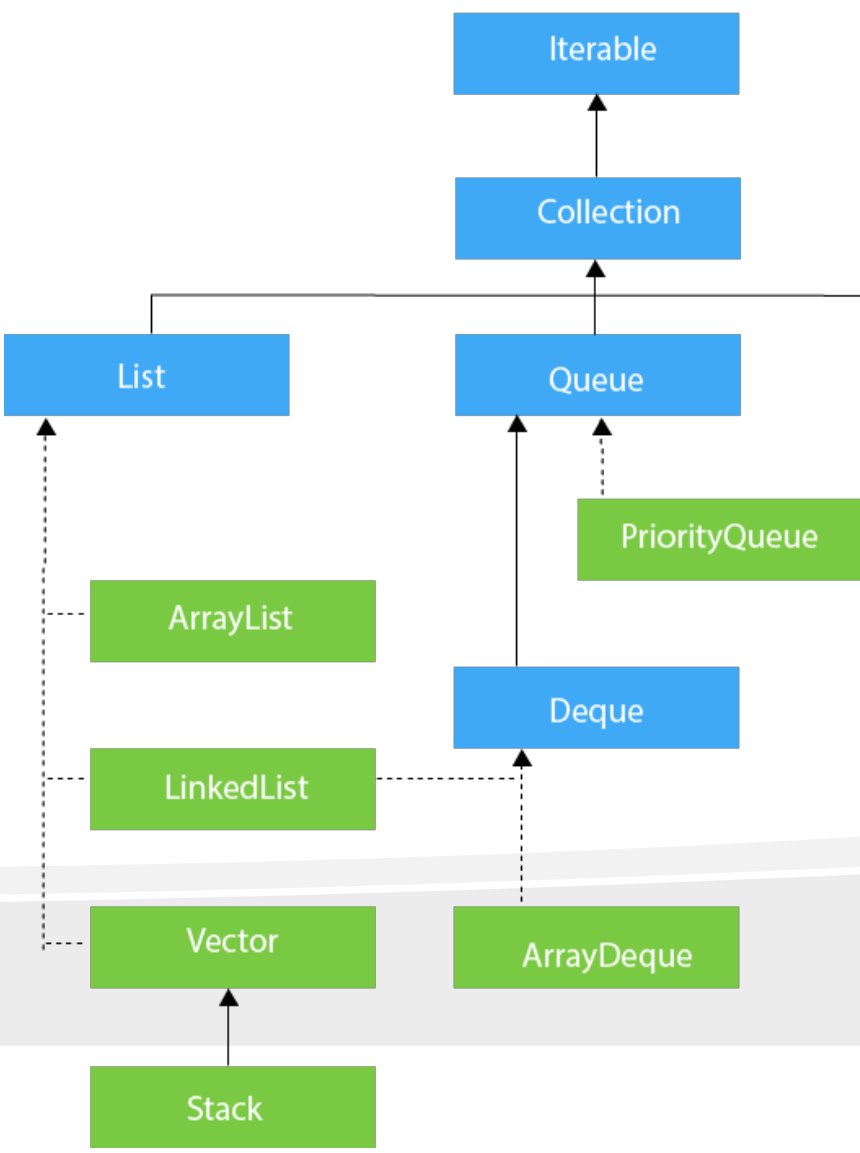
```
// The CalendarDate class stores information about a s
   date (month and day but no year).

public class CalendarDate implements Comparable<Calenda
    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;
    }
}
```

Review: Example Client Program

```
// Short program that creates a list of the birthdays of the
// 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
        22));   dates.add(new CalendarDate(10, 30));  dates.ad
        CalendarDate(4, 13));   dates.add(new CalendarDate(3,
        dates.add(new CalendarDate(4, 28));

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

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 D


Iterable
Collection
List
Queue
PriorityQueue
ArrayList
Deque
LinkedList
Vector
ArrayDeque
Stack

Internals of `ArrayList`

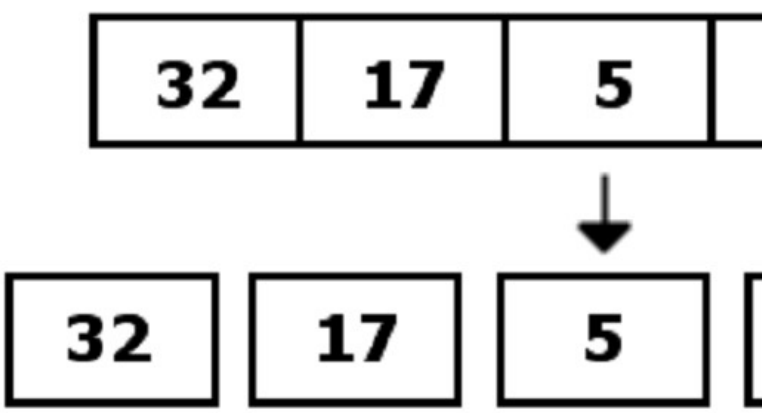
- It is internally stored in an array. So, it has a certa
- Capacity: size of the array used to store the elements i
- It is always at least as large as the list size
- As elements are added to an `ArrayList`, its cap
- Random access to elements
- `set/get` an element to/at specific index position has c
- To `add` at the end of the `ArrayList` it takes **O(1)**
- Remove, or add at a specified index operations ha

The underlying issue

- The elements of an `ArrayList` are too tightly att

them - Can we break the element storage apart into

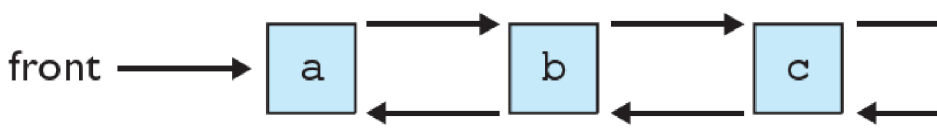
structure?



Linked list

- **Linked list** is a list implemented using a linked s
- Each value is stored in a small object called a **node**, v 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 `Arra`
- It also implements the `List` interface, so it offers the

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

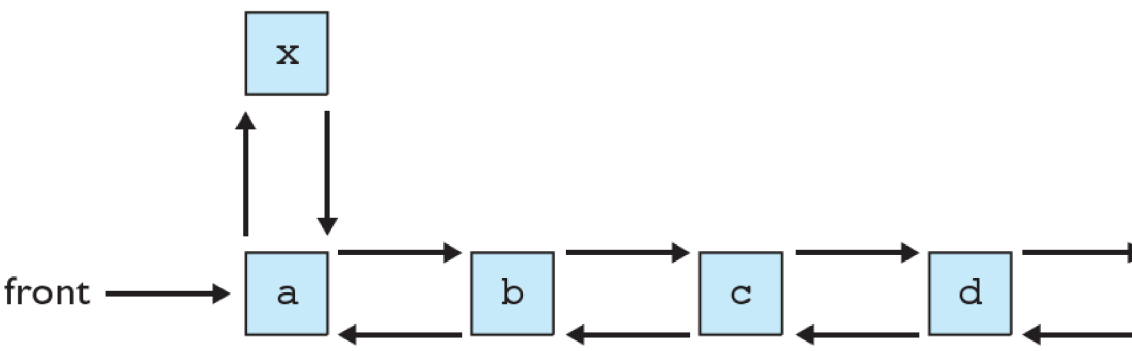
```
words.add("goodbye"); words.add("this");
words.add("that");
```

- Advantage: elements are added/removed at/from
- There's no shifting, we just create a new node object a

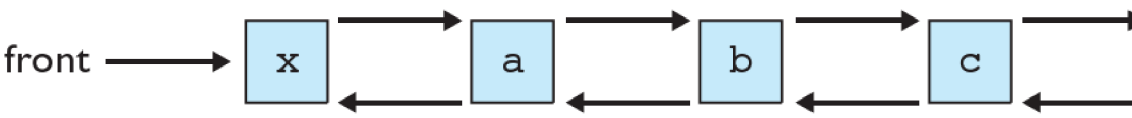
Adding elements to the list

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

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

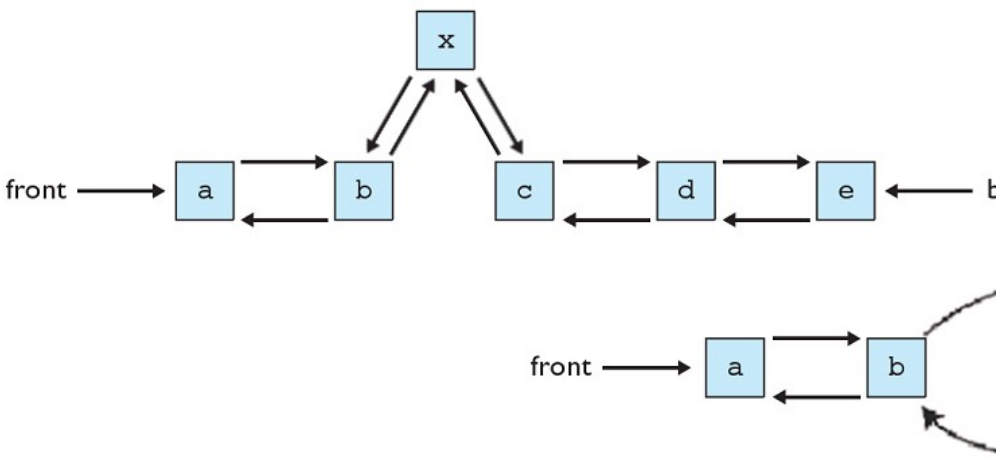


3. Change the front of the list to point to the new node.



Linked list performance

- To add, remove, get a value at a given index:
- The list must advance through the list to the node ju
- For example, to add a new value to the list, the list c
  node links to the proper index, and attaches it to the
- This is very fast when adding to the front or back of
  references to these places), but slow elsewhere


x
front
a
b
c
d
e
front
a
b

Linked List Implementation – Inne

```
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;
            }
   }
}
```

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`) eve times
- This code will take long time to run for large data siz
- **Sequential access is** slower than the **random acce**

The problem of position

- The code on the previous slide is wasteful because
  - Every call to `get` has to re-traverse the list
- It would be much better if we could somehow keep
  we looped through it
- Java uses special objects to represent a position o
- These objects are called **iterators**

`Collection` Interface

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

`Iterator<E>` Interface

- Defines three fundamental methods
- `E next()` returns the next element. If there are no m

`NoSuchElementExecption`

- `boolean hasNext()` returns `true` if there is anoth
- `void remove()` removes the last element returned b a call to `next`)
- These three methods provide access to the conte
- An `Iterator` knows position within a collection
- Each call to `next()` "reads" an element from the
- Then you can use it or remove it

Iterator

- The easiest way to cycle through the elements in a

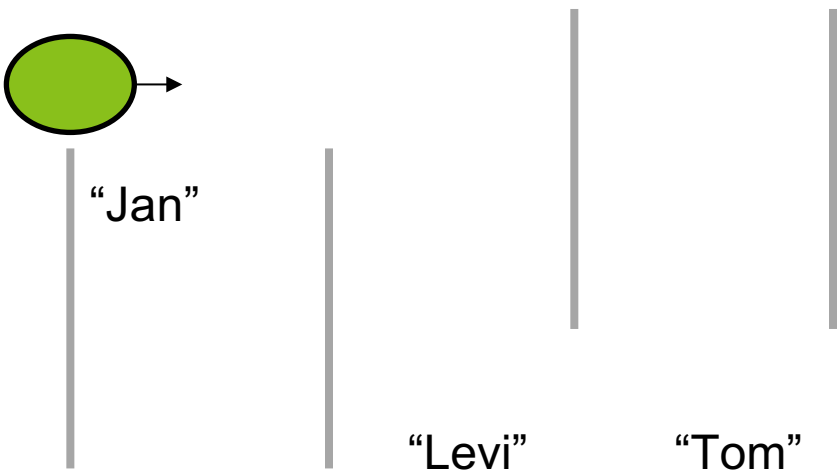
```
Iterator<type> iter = collection.it
while (iter.hasNext()){ type next
iter.next(); // do something wi
nextElement }
```

Iterator

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

          // Create an iterator for the
          Iterator<String> iter = names

          // Use the iterator to visit
          ( iter.hasNext() )
                    System.out.println(
     }
}
```

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;
```

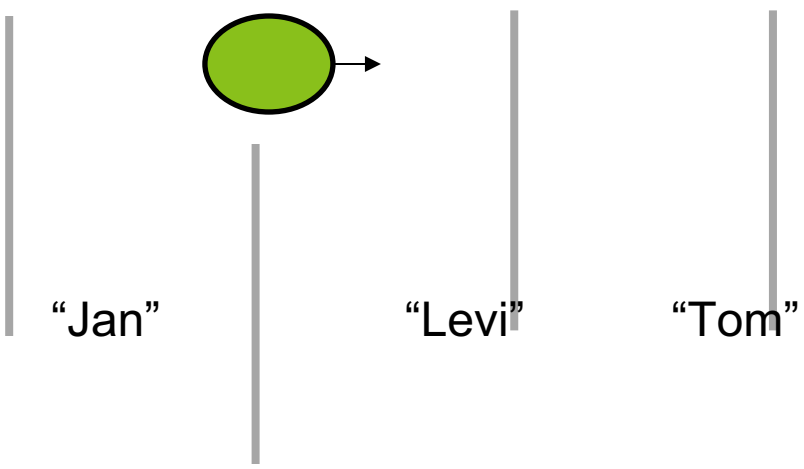


Iterator Position

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

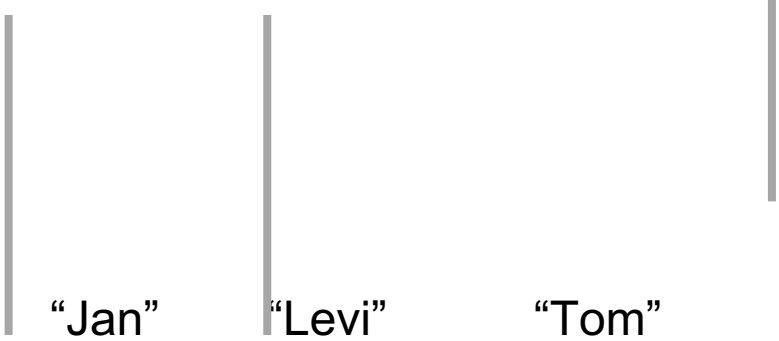
first call to `next` moves itera

Iterator Position



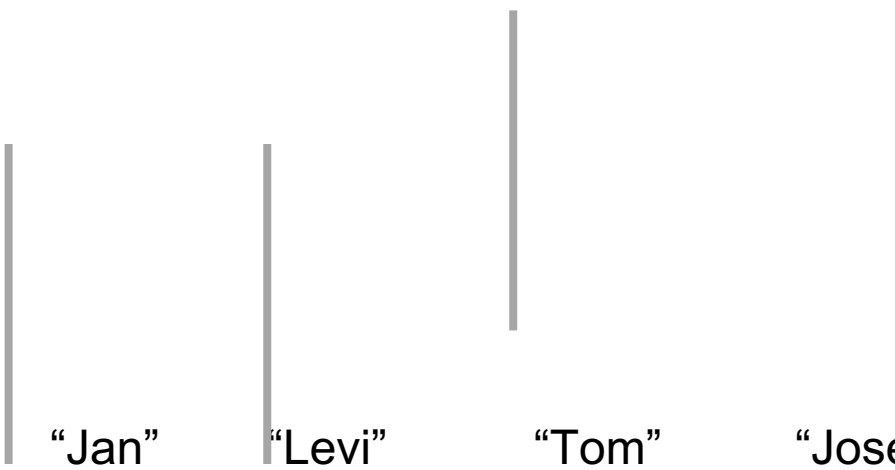

```
while(
     it.hasNext() ) { i++;
     System.out.println( it.next() );
}
// when i == 2, prints out Levi
```

Iterator Position



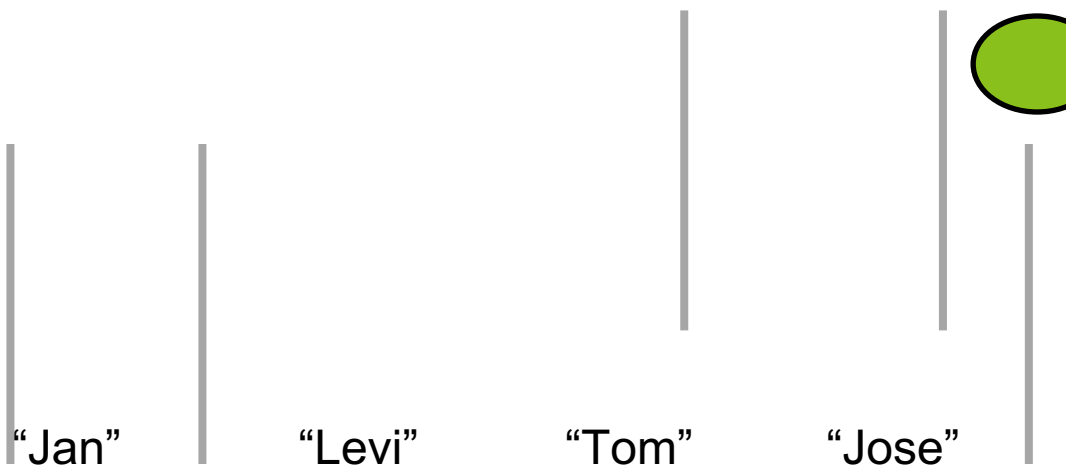

```
while( it.hasNext() ) {
    i++;
    System.out.println( it.next() );
}
// when i == 3, prints out Tom
```

Iterator Position



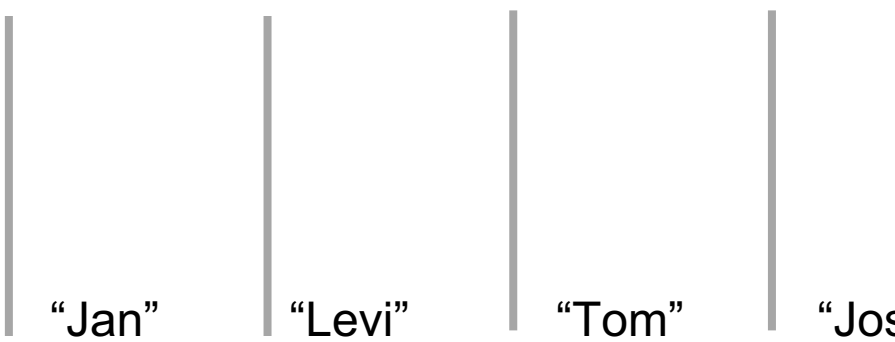

```
while( it.hasNext() ) {
    i++;
    System.out.println( it.next() );
}
// when i == 4, prints out Jose
```

Iterator Position



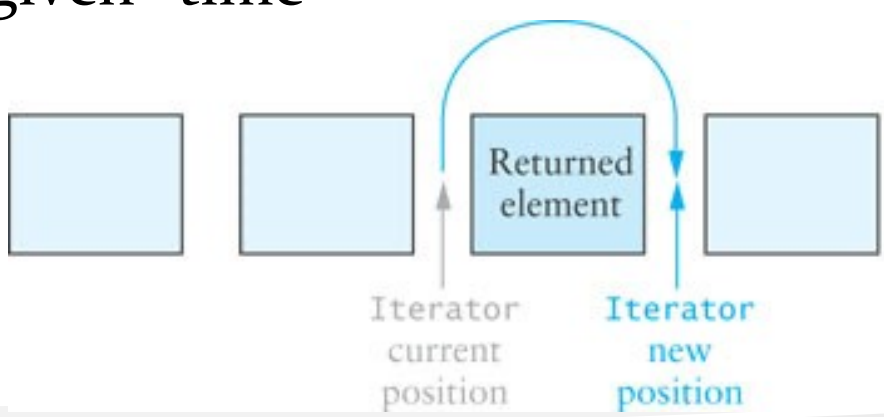

```
                                    it.hasNext()
    System.out.println( it.next() );
}
// call to hasNext returns false
// while loop stops
```

Iterator Position



Iterator Position

- An `Iterator` is conceptually *b*
  does notrefer to a particula
  given time



Fixing the slow LL idiom

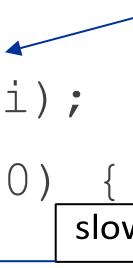
```
// print every element of the list for
(int i = 0; i < list.size(); i++) {
    Object element = list.get(i);
    System.out.println(i + ": " + element);
}
```

```
// 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 ele
   }
}
```



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 cha

```
// removes all strings of even length
public static void removeEvenLength(L
    Iterator<String> i = list.iterato
    {
        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`

- 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 elem

- 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

Concurrent modification exceptio

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

- While you are still iterating, you cannot call any list's contents
- The code crashes with a `ConcurrentModificatio`
- It is okay to call a method on the **iterator itself** that m

- `get` and `remove` loops ONLY (not `set/add` opera

The `ListIterator<E>` interf

- Extends the `Iterator` interface

- The `LinkedList` class implements the `List<E>` i

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

- Methods in the `ListIterator` interface:
- `add, hasNext, hasPrevious, next, previo`
  `remove, set`

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

- The following static methods in the `Collections`
- Example: `Collections.replaceAll(list, "hello", "`

Method name	
`binarySearch(`*list*`, `*value*`)`	searches a sorted list fo
`copy(`*dest*`, `*source*`)`	copies all elements fron

`fill(`*list*`, `*value*`)`	replaces all values in th
`max(`*list*`)`	returns largest value in
`min(`*list*`)`	returns smallest value
`replaceAll(`*list*`, `*oldValue*`, `*newValue*`)`	replaces all occurrences
`reverse(`*list*`)`	reverses the order of e
`rotate(`*list*`, `*distance*`)`	shifts every element's i
`sort(`*list*`)`	places the list's elemen
`swap(`*list*`, `*index1*`, `*index2*`)`	switches element value