COLLECTION FRAMEWORK

Java 1.2 introduced an extensive set of interfaces and classes that act as containers (of Object).

- Found in package *java.util*
- Replaces Vector, Hashtable, Stack of Java 1.1.

Limitations of Arrays

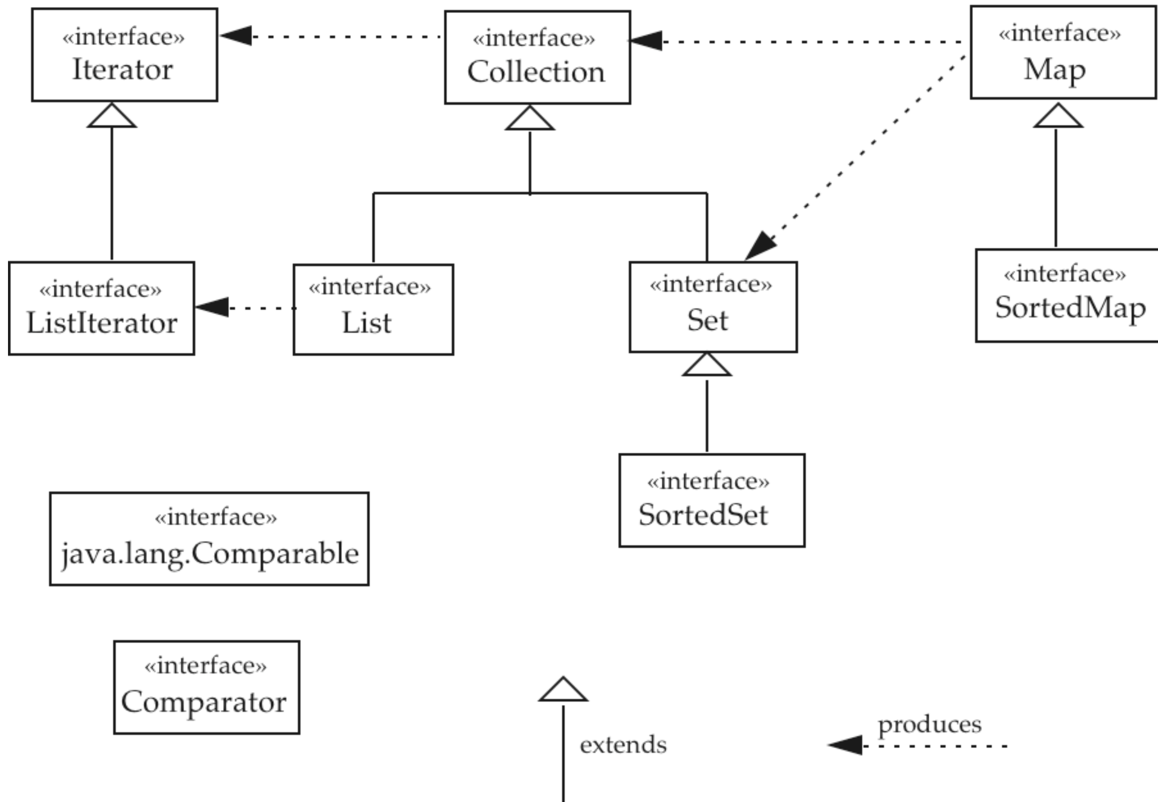
- Need to estimate size of array since once it has been created, it cannot grow or shrink.
- Want containers that can grow dynamically and that are efficient to use.

The collection or container framework has interfaces that specify the syntax of operations to be implemented in various classes.

When container objects are created, they will be referred to through interface variables, which allow the implementation to be changed without changing the code that uses it.

The interfaces are organized in terms of the intended purposes of the objects that will be created.

Container Interface Hierachy



Java has no direct implementation of Collection currently.

 Copyright 2006 by Ken Slonneger

Collection Interface

A Collection is a group of elements of type Object.

Basic operations: add elements, remove elements, and look at elements with no assumptions about order or duplicates.

All of the methods in the interface are **public** and **abstract**.

```
public interface Collection
{
// Basic Operations
    boolean add(Object ob);              // return true if a
    boolean remove(Object ob);           // change is made
    boolean contains(Object ob);
    int size();
    boolean isEmpty();
    Iterator iterator();      // an Interator object produces all
                              // the elements in the collection

// Bulk Operations
    void clear();
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    boolean containsAll(Collection c);

// Array Operations
    Object [] toArray();
    Object [] toArray(Object [] oa);     // type of result is
                                         // the type of oa
}
```

List Interface

This subinterface of Collection assumes the elements have ordered positions, starting at position zero.

It specifies additional behavior.

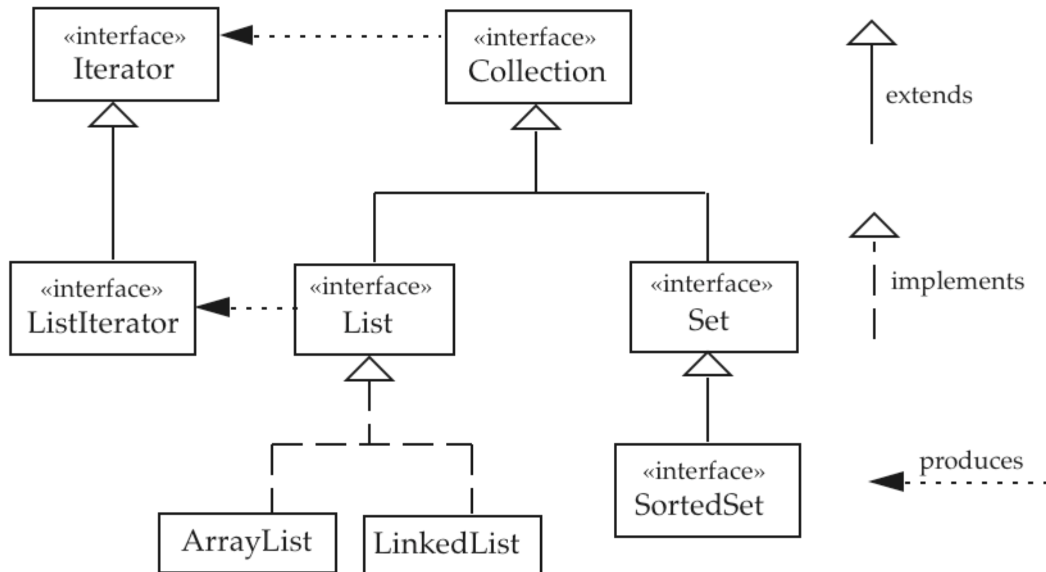
```
public interface List extends Collection
{
// Positional access (p is a position)
    Object get(int p);
    Object set(int p, Object ob);    // returns old component
    void add(int p, Object ob);
    Object remove(int p);            // returns old component
    boolean addAll(int p, Collection c);

// Searching
    int indexOf(Object ob);
    int lastIndexOf(Object ob);

// Range view
    List subList(int p1, int p2);

// Another iterator
    ListIterator listIterator();      // ListIterator allows backward
    ListIterator listIterator(int p); // as well as forward iteration
}
```

The List interface is implemented by two concrete classes:

ArrayList	contains a “dynamic” array of Objects
LinkedList	contains a linked list of nodes such that each holds one Object.

 Copyright 2006 by Ken Slonneger

Implementing List



ArrayList Class

An ArrayList object encapsulates an array of Object that can be resized (actually, by allocating a new array).

Constructors

ArrayList()	// an empty list
ArrayList(Collection c)	// a list with c's elements
ArrayList(**int** cap)	// an empty list with capacity cap

Example

```
List aList = new ArrayList();
aList.add("Sun");       aList.add("Mon");
aList.add("Tue");       aList.add("Wed");
aList.add("Thu");       aList.add("Fri");
aList.add("Sat");
```

Now the array inside the object aList holds seven String objects (actually references to the Strings).

Retrieving the Elements

1. A **for** loop with "subscripting".

```
for (int k=0; k<aList.size(); k++)
        System.out.println(aList.get(k));
```

2. An Iterator object

   The interface Iterator has three methods

```
boolean hasNext();     // true if more elements to visit
Object next();         // returns "next" element
void remove();         // removes last element visited
```

   An Iterator guarantees that each element will be visited exactly once, but the order is unspecified.

```
Iterator elements = aList.iterator();
while (elements.hasNext())
{
        Object ob = elements.next();
        System.out.println(ob);
}
```

   The Object may need to be downcast to perform specialized operations on it, say to get the length of a String.

 Copyright 2006 by Ken Slonneger

Example

- Read floating-point numbers and place them into an ArrayList.
- Find the mean (average) of the numbers.
- Remove the duplicates from the ArrayList.
- Make an alias of the list and change an element.

The program does its own tokenizing of the numbers in the input stream using the instance method *substring* from String.

```
import java.io.*;
import java.util.*;

public class MakeAL
{
    static List readNums()
    {
        List list = new ArrayList();          // Point 1

        BufferedReader brdr =
                new BufferedReader(
                        new InputStreamReader(System.in));
        try
        {   String str = brdr.readLine();
            while (str != null)
            {
                int pos = 0;
                while  (pos<str.length())
                {
                    int next = str.indexOf('  ', pos);
                    if (next==-1) next = str.length();
                    String word = str.substring(pos, next);
```

```
                  if (word.length()>0)
                  {
                       Double d = Double.valueOf(word);
                       list.add(d);
                  }

                  pos =next+1;
             }
             str = brdr.readLine();
        }
    }
    catch (IOException e)  {  }
    return list;
}
```

Trace

str =

9	8	7		6	5	.	4			3	2
0	1	2	3	4	5	6	7	8	9	10	11

pos	next	word	d
0	3	"987"	987.0
4	8	"65.4"	65.4
9	9	""	none
10	-1		
	12	"32"	32.0
13			

MakeAL Continued

```
static double findMean(List lst)
{
    double sum = 0;
    Iterator it = lst.iterator();
    while (it.hasNext())
    {
        Double d = (Double)it.next();
        sum = sum + d.doubleValue();
    }
    if (lst.size()>0) return sum/lst.size();
    else return 0.0;
}

static void printList(List lst)
{
    for (int k=0; k<lst.size(); k++)
        System.out.println("Number " + k + ": " +
                          ((Double)lst.get(k)).doubleValue());
}

static List removeDups(List lst)
{
    List newList = new ArrayList();          // Point 2
    for (Iterator it=lst.iterator(); it.hasNext();    )
    {
        Object ob = it.next();
        if (!newList.contains(ob))   newList.add(ob);
    }
    return newList;
}
```

```
    public static void main(String [] args)
    {
        System.out.println("Enter real numbers separated "
                              + " by spaces, terminated by ctrl-d");
        List list1 = readNums();
        System.out.println("List 1:");
        printList(list1);
        double mean = findMean(list1);
        System.out.println("Mean = " + mean);
        List list2 = removeDups(list1);
        System.out.println("List 2:");
        printList(list2);
        List list3 = list2;
        if (list2.size() > 2)
            list2.set(2, new Double(-999.99));
        System.out.println("List 3:");
        printList(list3);
    }
}
```

Using Scanner

```
    static List readNums()
    {
        List list = new ArrayList();              // Point 1
        Scanner scan = new Scanner(System.in);
        while (scan.hasNextDouble())
        {
            Double d = new Double(scan.nextDouble()};
            list.add(d);
        }
        return list;
    }
```

 Copyright 2006 by Ken Slonneger

Output

Enter real numbers separated by spaces, terminated by ctrl-d
123 45.65 33 88.7 123
987 44.9 33 123

List 1:
Number 0: 123.0
Number 1: 45.65
Number 2: 33.0
Number 3: 88.7
Number 4: 123.0
Number 5: 987.0
Number 6: 44.9
Number 7: 33.0
Number 8: 123.0
Mean = 177.91666666666666
List 2:
Number 0: 123.0
Number 1: 45.65
Number 2: 33.0
Number 3: 88.7
Number 4: 987.0
Number 5: 44.9
List 3:
Number 0: 123.0
Number 1: 45.65
Number 2: -999.99
Number 3: 88.7
Number 4: 987.0
Number 5: 44.9

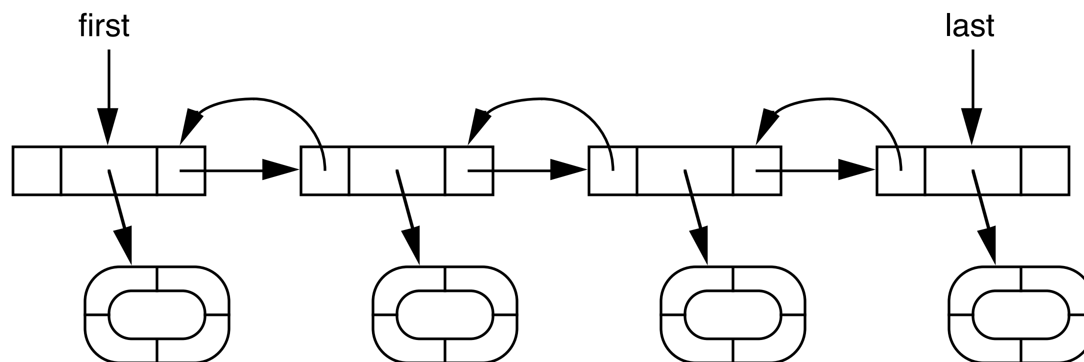
LinkedList Class

LinkedList implements List.

A LinkedList object is a sequence of nodes with

1. A reference to the first node.
2. A reference to the last node.
3. Each node has
    - a reference to its predecessor (**null** if first node).
    - a reference to its successor (**null** if last node).
    - a reference to an Object, the datum stored in the node.

A LinkedList



Advantages of ArrayList over LinkedList

- An element of an ArrayList can be accessed directly (an address calculation).
- To access an element in a LinkedList, the list must be traversed to the position of the element.

Advantages of LinkedList over ArrayList

- An insertion or deletion in a LinkedList can be made by changing just a few references.
- An insertion or deletion in an ArrayList requires moving all following elements up or down in the array.

LinkedList Constructors

LinkedList()	// an empty linked list
LinkedList(Collection c)	// a linked list with c's elements

Change MakeAL into MakeLL

Since the variables *list* and *newList* are declared of type List, they can be assigned either an ArrayList or a LinkedList.

Make two changes in the code of MakeAL:

```
static List readNums()
{
    List list = new LinkedList();          // Point 1


static List removeDups(List lst)
{
    List newList = new LinkedList();       // Point 2
```

The behavior of the program is unchanged by this alteration.

Such a change may, however, affect performance (efficiency).

Testing

These operations show the best and worst performance using ArrayList and LinkedList.

```
List list = new ArrayList();      or    List list = new LinkedList();
```

```
for (int k=1; k<=50000; k++)
      list.add(0, new Integer(k));
```

ArrayList: 48.576 seconds

LinkedList: 1.115 seconds

```
int sum = 0;
for (int k=1; k<=50000; k++)
      sum = sum + ((Integer)list.get(25000)).intValue();
```

ArrayList: 0.213 seconds

LinkedList: 184.252 seconds

 Copyright 2006 by Ken Slonneger

Indexed Lists

array
```
Integer [] a = new Integer [50];
```

List
```
List list = new ArrayList();
```

String
```
String s = "to be or not to be";
```

StringBuffer
```
StringBuffer sb = new StringBuffer(s);
```

Comparison of Operations

m = a[k];	m = list.get(k);	c = s.charAt(k);
a[k] = w;	list.set(k, w);	sb.setCharAt(k, w);
a.length	list.size()	s.length()
if (a.length==0) { … }	**if** (list.isEmpty()) { … }	**if** (s.length()==0) { … }

Collections Contain Objects

- When objects are inserted into a collection, they are automatically upcasted to type Object.
- Since items in a collection are viewed to be of type Object, they must be downcast (explicitly cast) to their actual types to be used.

We must be certain of the type of the Objects before downcasting; otherwise get a ClassCastException

Problem

- Collections are inherently unsafe since objects of many different types may be entered into them.
- Downcasting can therefore be unpredictable.

Solutions

- Provide a method that allows objects of only one type to be added to a collection, making it homogeneous. Type checking done by parameter to method.

  For example,

```
static void addDomino(Domino d)
{
	lst.add(d);
}
```

  or

- Test every Object from a collection using **instanceof**.

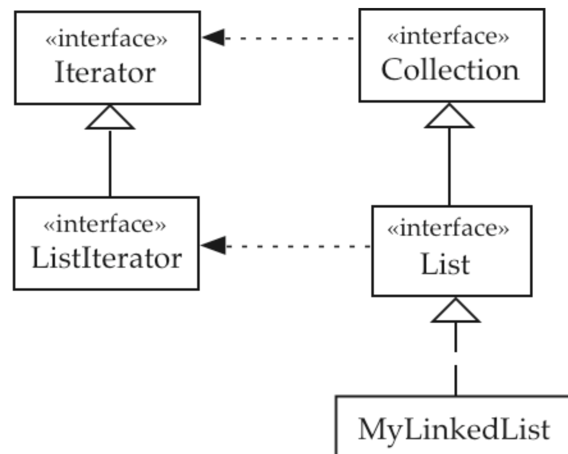
 Copyright 2006 by Ken Slonneger

Implement List Using a Linked List

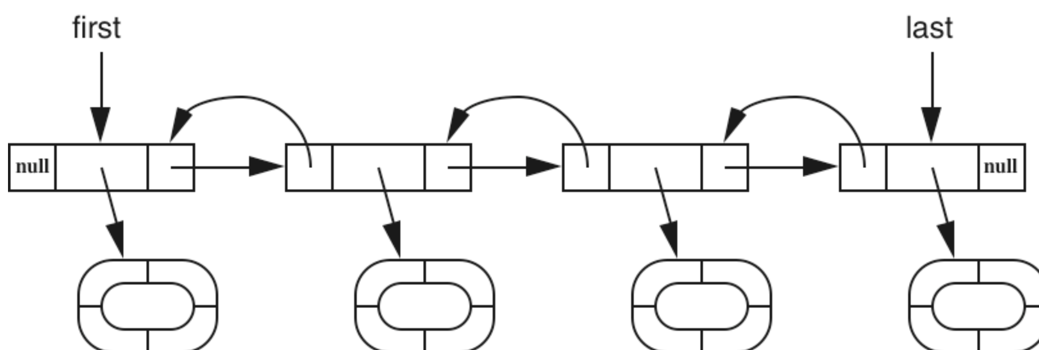
Java implements List with the concrete class LinkedList.

We want to duplicate this effort to illustrate how linked structures are realized in Java.

Call our implementation MyLinkedList.



Structure of a Linked List Implementation



The four objects in this list are referred to by object structures that we call nodes. Each node contains three references.

Node Class

The node objects are defined using an inner class.

Each node object has three fields.

```
public class MyLinkedList implements List
{
    private class Node
    {
        Node prev, next;
        Object item;

        Node(Node p, Object ob, Node n)
        {
            prev = p;  item = ob; next = n;
        }
    }
```

A MyLinkedList object contains three instance variables, the references to the first and last nodes of the list and a variable to hold the size of the list.

```
    private Node first;
    private Node last;
    private int size;
```

Conditions for an Empty List

```
        first == null
        last == null
        size == 0
```

 Copyright 2006 by Ken Slonneger

Constructors for MyLinked List

```
public MyLinkedList()
{
    first = null; last = null; size = 0;
}

public MyLinkedList(Collection c)
{
    this();                         // redundant
    Iterator it = c.iterator();
    while (it.hasNext())
        add(it.next());         // need to implement add
}
```

Easy Instance Methods

```
public int size()
{
    return size;
}

public boolean isEmpty()
{
    return size==0;
}

public void clear()
{
    first = null;
    last = null;
    size = 0;
}
```

Method add

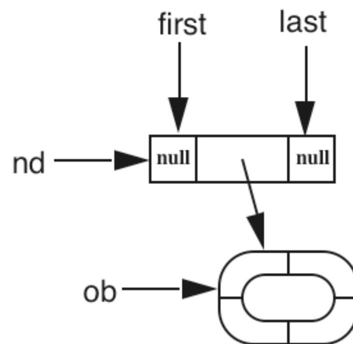
Most of the behavior of a Collection can be defined in terms of the basic methods: *add*, *contains*, *remove*, and *iterator*.

We require that the *add* method only accepts non-null objects.

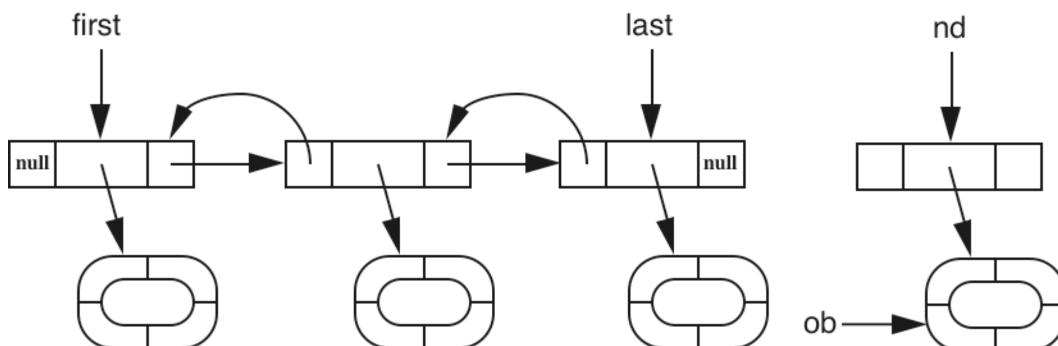
It increments the *size* variable when an object is added.

Two cases

1. Add the object to an empty list.



2. Add the object at the end of a nonempty list.




```
public boolean add(Object ob)
{
    if (ob == null)
        throw new IllegalArgumentException();
    size++;
```

 Copyright 2006 by Ken Slonneger

```
        if (first == null)            // add to an empty list
        {
            Node nd = new Node(null, ob, null);
            first = nd;
            last = nd;
        }
        else                          // add to a nonempty list
        {
            Node nd = new Node(last, ob, null);
            last.next = nd;
            last = nd;
        }
        return true;
    }
```

Method *contains*

The *contains* method requires that we search the linked list for a particular object.

We search by moving through the nodes looking for a match with the object.

The variable *spot* refers to the node currently being tested.

If the search fails, *spot* will get the value **null**.

Note the need for a conditional *and* operation in the loop.

```
    public boolean contains(Object ob)
    {
        Node spot = first;
        while (spot != null && !ob.equals(spot.item))    // search
            spot = spot.next;
        return (spot != null);             // true if spot is not null
    }
```

Method *remove*

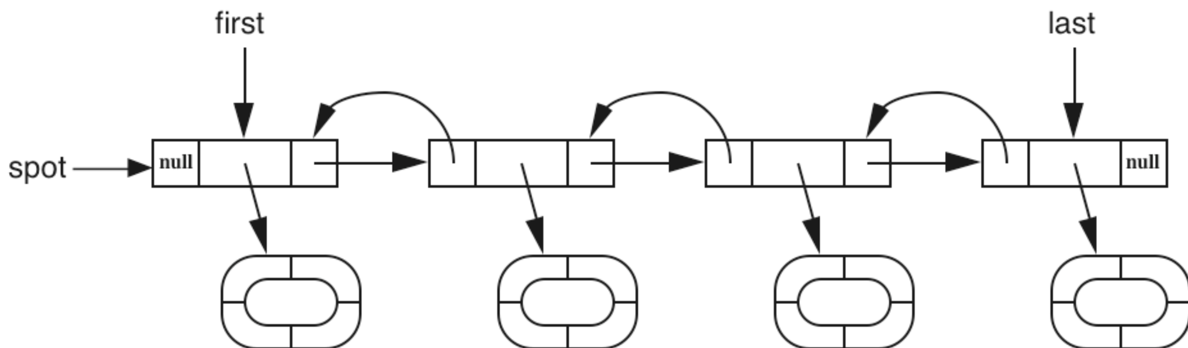
We search the list for the object to be removed.

If it is not found, return false.

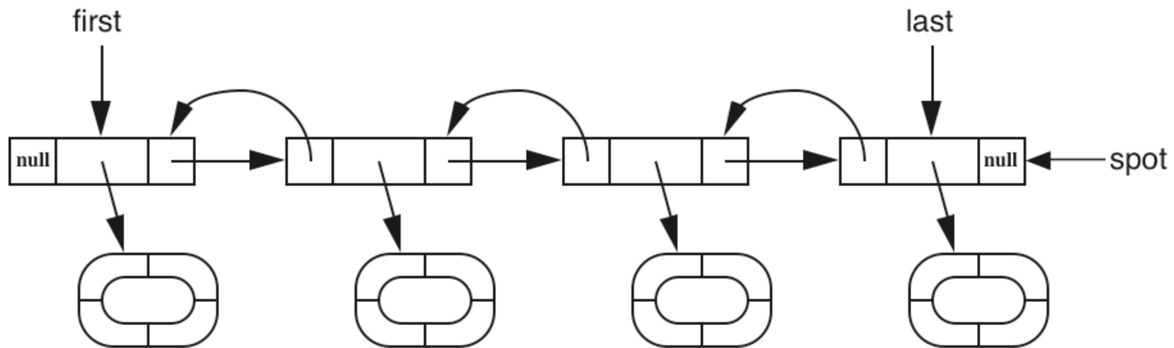
If the object is found, decrement *size,* remove the node with the object, and return true.

Three cases

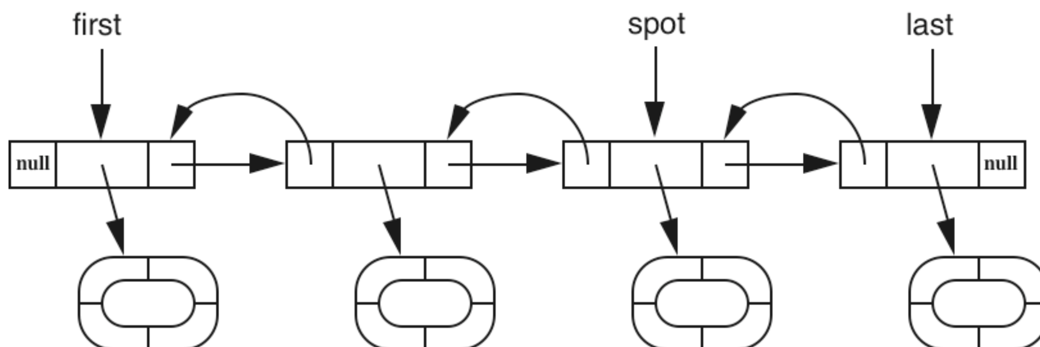
1. Object ob is the first item in the list.



2. Object ob is the last item in the list.



3. Object ob is somewhere between the first and last items.



 Copyright 2006 by Ken Slonneger

```
public boolean remove(Object ob)
{
    Node spot = first;
    while (spot != null && !ob.equals(spot.item))    // search
        spot = spot.next;

    if (spot == null)
        return false;

    size--;
    if (spot == first)
    {
        first = first.next;
        if (first == null)
            last = null;       // an empty list
        else
            first.prev = null;
    }
    else if (spot == last)
    {
        last = last.prev;     // list not empty
        last.next = null;
    }
    else
    {   spot.prev.next = spot.next;
        spot.next.prev = spot.prev;
    }

    return true;
}
```

Building an Iterator

The *iterator* method needs to return an object from a class that implements the interface Iterator.

```
public Iterator iterator()
{
    return new MyIterator();    // need to define this class
}
```

MyIterator Class

Implements the Iterator interface.

```
public interface Iterator
{
    boolean hasNext();    // true if more elements to visit
    Object next();        // returns "next" element
    void remove();        // removes last element visited
}
```

Make it an inner class.

Use two instance variables that refer to

1. the node of the most recently returned object (*lst*)
2. the node of the next object to be returned (*nxt*)

Conditions

nxt == **null** means no more items to visit

lst == **null** means no last item visited because
a) we are at beginning of traversal or
b) the last item was removed.

An item must be visited before it can be removed, and it can only be removed once.

 Copyright 2006 by Ken Slonneger

```
private class MyIterator implements Iterator
{
    Node lst = null;   // position of last item visited
    Node nxt = first;  // position of next item to be visited

    public boolean hasNext()
    {
        return (nxt != null);      // true if nxt is not null
    }

    public Object next()
    {
        if (nxt == null)
            throw new NoSuchElementException();
        lst = nxt;               // lst is node being visited
        nxt = nxt.next;          // nxt is next node to be visited
        return lst.item;
    }
```

// Compare the Iterator *remove* method with the *remove*
// method in MyLinkedList.

// The variable *lst* refers to the node being removed.

```
    public void remove()
    {
        if (lst == null)
            throw new IllegalStateException();
```

```
            size--;          // lst refers to node to be removed
            if (size == 0)
                first = last = null;
            if (lst == first)
            {
                first = first.next;
                first.prev = null;
            }
            else if (lst == last)
            {
                last = last.prev;
                last.next = null;
            }
            else
            {   lst.prev.next = lst.next;
                lst.next.prev = lst.prev;
            }
            lst = null;          // cannot remove this item again
        }
    }
```

Remaining Collection Methods

Most of the remaining Collection methods can be defined in terms of the basic methods.

Here are two examples.

```
    public boolean addAll(Collection c)
    {
        boolean change = false;
        Iterator it = c.iterator();
```

 Copyright 2006 by Ken Slonneger

```
        while (it.hasNext())
            if (add(it.next()))
                change = true;
        return change;
    }

    public boolean removeAll(Collection c)
    {
        boolean change = false;
        Iterator it = c.iterator();
        while (it.hasNext())
        {
            Object ob = it.next();
            while (contains(ob))
            {
                remove(ob);
                change = true;
            }
        }
        return change;
    }
```

Exercise: Implement *toArray(), retainAll(Collection c)*, and *containsAll(Collection c)*.

The other *toArray* method is more difficult to implement, requiring methods from the java.lang.reflect package.

Remaining Methods

Several methods from Object must be implemented if we want the List objects to have correct behavior.

```
    public boolean equals(Object other)
    public int hashCode()
    public String toString()
```

```
public String toString()
{
    String result = "[";
    Iterator it = iterator();
    if (it.hasNext())
        result = result + it.next();
    while (it.hasNext())
        result = result + ", " + it.next();
    return result + "]";
}
```

Exercise: Write the methods *equals(Object ob)* and *hashCode()*.

That leaves the methods in the List interface, an extension of Collection.

Most of these methods require that we find an index position in the list. The simplest one is the *get* method.

```
public Object get(int index)
{
    if (index<0 || index>=size)
        throw new IndexOutOfBoundsException();
    Node spot = first;
    while (index > 0)
    {
        spot = spot.next;
        index--;
    }
    return spot.item;
}
```

 Copyright 2006 by Ken Slonneger

The *add(int index, Object ob)* and *remove(int index)* methods are similar to the Collection methods once the position of the index is found.

The *addAll(int index, Collection c)* method cannot simply use the existing method *add(int index, Object ob)* because we would have to index the list for each element of the Collection, and indexing in a linked list means a linear search. Therefore, this *addAll* must be written from scratch.

Exercise: Implement the List methods *set(int index, Object ob)*, *indexOf(Object ob)*, *lastIndexOf(Object ob)*, add*(int index, Object ob)*, and *remove(int index)*.

Method ***subList***

The *subList(int from, int to)* method is the trickiest.

The following strategy seems to work:

- Write a private constructor for MyLinkedList that takes a Node (*first*) and two integers (*from* and *to*) as parameters.
- Inside this constructor find the nodes corresponding to the *from* position and the spot in front of the *to* position and initialize the new MyLinkedList object using these nodes to set *first* and *last*.
- In subList, use the private constructor to return a new MyLinkedList object that identifies the sublist.

The documentation states that the List returned by this method becomes “undefined” if the original List is modified structurally in any way.

ListIterator Methods

A ListIterator object will be similar to an Iterator object, but a little more complicated. I did not implement the two methods that produce a ListIterator object.

To satisfy the compiler, the missing methods just throw an exception.

```
ListIterator listIterator()
{
    throw new UnsupportedOperationException();
}

ListIterator listIterator(int index)
{
    throw new UnsupportedOperationException();
}
```

Other LinkedList Methods

LinkedList implements six other methods that occur in no interface.

```
public Object getFirst()
public Object getLast()
public Object removeFirst()
public Object removeLast()
public void addFirst(Object ob)
public void addLast(Object ob)
```

The first four methods throw NoSuchElementException if the list is empty.

Exercise: Implement these six methods.

 Copyright 2006 by Ken Slonneger

Set Interface

A Set is a Collection with no duplicate elements.

As with mathematical sets, the elements are not assumed to be in any particular order.

The Set interface extends Collection but has no additional methods.

HashSet Class

HashSet implements Set using a hash table.

Every object in Java has a number associated with it.

The class Object contains an instance method

public int hashCode()

for producing the numbers, ideally evenly distributed over the range of the function.

Most classes override *hashCode* using properties of the class to generate hash numbers that are scattered.

Basic Property

If ob1.equals(ob2), then ob1.hashCode() == ob2.hashCode().

A hash table stores objects by computing their hash values and using those values to index into the table (an array).

If the table has *m* slots, numbered 0 to m-1, an item with hash code *h* is placed at position |h % m|.

Hash Function

hash(key) = Math.abs(key.hashCode() % m)

Collision Resolution

A collision occurs if two different keys, called *synonyms*, hash to the same position in the hash table:

$$hash(key_1) = hash(key_2) \text{ for } key_1 \neq key_2$$

Separate Chaining

Collisions are handled by maintaining a linked list, called a bucket, for each hash value.

Synonyms are linked together on one of these lists.

Example

Consider a collection of elements that are animal names (Strings).

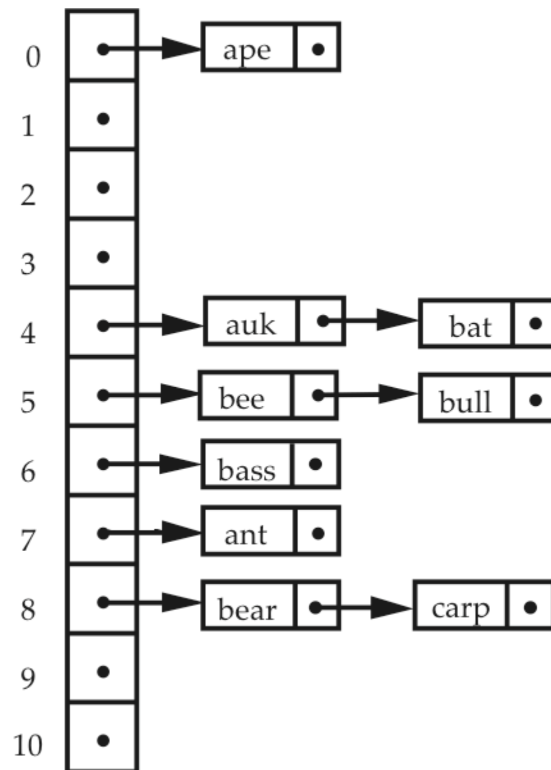
Suppose that the tableSize is 11 and that the hash value is calculated using:

$$hash(key) = |key.hashCode() \% 11|$$

	Hash		Hash		Hash
ant	7	bat	4	carp	8
ape	0	bear	8	cat	9
auk	4	bee	5	cod	5
bass	6	bull	5	eel	5

Insert the first 9 animals into a hash table with 11 slots.

 Copyright 2006 by Ken Slonneger



Note that the number of keys stored in a hash table with separate chaining is limited only by the dynamic allocation heap (where objects are stored).

Furthermore, deletions are simply linked list deletions involving the modification of one reference.

However, searching slows down when the linked lists get very long.

Load Factor = (number of elements)/(number of slots)

Default load factor = 0.75.

When load factor is greater than 0.75, table is reorganized with a larger array.

Algorithm *hashCode* for String

A *class* method since we cannot get inside the String class.

```
static int hc(String s)   // early versions of Java used this code
{
    char [] val = s.toCharArray();
    int len = val.length;
    int h = 0;
    if (len<16)
        for (int k=0; k<len; k++)         // Horner's method for
            h = 37*h + val[k];            // evaluating a polynomial
    else
    {   int skip = len/8;
        for (int k=0; k<len; k=k+skip)
            h = 39*h + val[k];            // What about overflow?
    }
    return h;
}
```

Sample Hash Codes

"herky" ⇒ 200188843

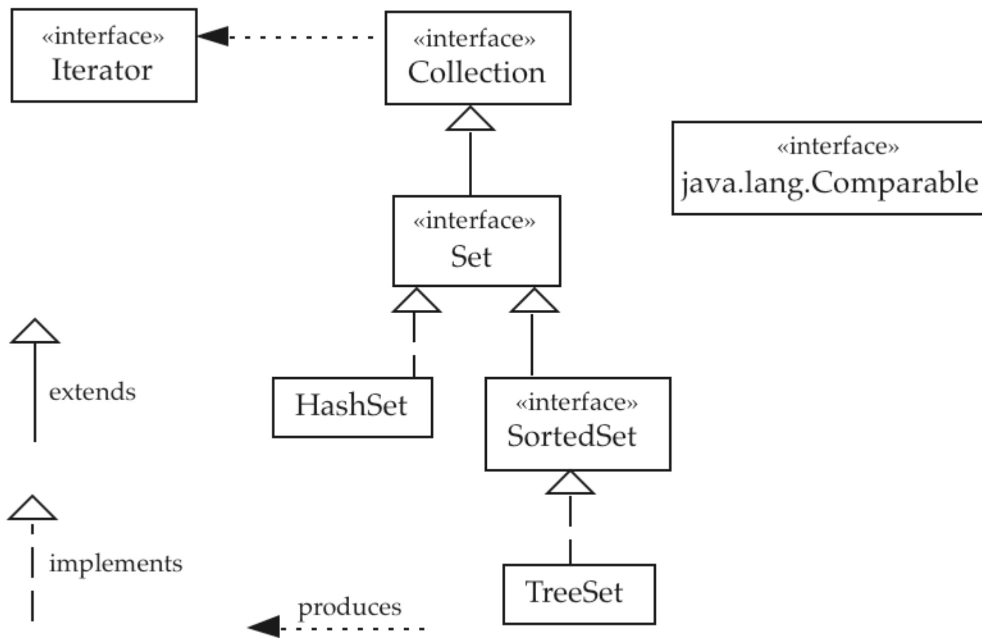
"to be or not to be that is the question" ⇒ -1383044710

Current Java (1.4)

```
for (int k=0; k<len; k++)       // Used for strings of
    h = 31*h + val[k];          // any length
```

 Copyright 2006 by Ken Slonneger

Set and SortedSet



HashSet Constructors

HashSet() // an empty hash table with default size

HashSet(Collection c) // a hash table with c's elements

HashSet(**int** sz) // an empty hash table with sz size

HashSet(**int** sz, **float** lf) // size sz and load factor lf

Another Version of *removeDups*

Duplicates can be removed by changing the Collection into a Set.

```
static List removeDups(List lst)
{
    Collection collect = new HashSet(lst);
    return new ArrayList(collect);
}
```

SortedSet Interface

This subinterface of Set expects to maintain the set in order according to an instance method found in the interface Comparable.

```
int compareTo(Object other)
```

All objects added to a SortedSet must belong to a class that implements the interface Comparable.

String and all the wrapper classes, except Boolean, implement Comparable.

Because of the ordering, SortedSet allows additional behavior.

```
public interface SortedSet extends Set
{
  Object first();
  Object last();
  SortedSet subSet(Object from, Object to);   // from ≤ x < to
  SortedSet headSet(Object to);               // first ≤ x < to
  SortedSet tailSet(Object from);             // from ≤ x ≤ last
  Comparator comparator();        // has a method to compare
}                                 // two objects
```


Copyright 2006 by Ken Slonneger

TreeSet Class

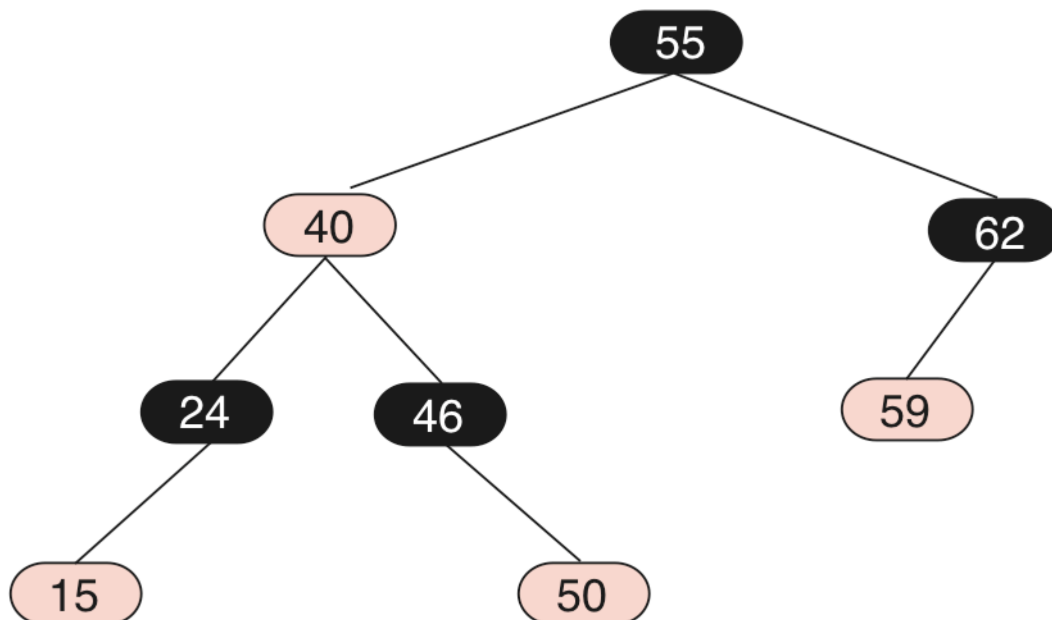
TreeSet implements SortedSet using a red-black tree.

A red-black tree is a binary search tree that has its nodes colored so that

- The root is black.
- No path from the root to a leaf has consecutive red nodes.
- All paths from the root to leaves have the same number of black nodes.

These conditions keep the tree balanced so that searching is efficient.

A Red-Black Tree



TreeSet Constructors

```
TreeSet()                   // an empty tree
TreeSet(Collection c)       // an ordered tree with c's elements
```

Ordering Domino

Put an ordering on Domino objects so they can be added to a SortedSet.

Ordering

- If the low values differ, use them to order the Domino objects.
- If the low values are the same, use the high values to order the objects.

We need Domino to implement the interface Comparable by providing code for compareTo().

When compareTo() is overridden, its definition must be consistent with equals(). Since Domino uses equals() from Object, it needs to be redefined also.

Possibilities

- Change the header for Domino and insert new methods into its class. This approach requires recompiling the Domino class.
- Subclass Domino with a class that implements Comparable and provides the needed code, including equals().
  Then Domino does not need to be touched.

We follow the second approach.

We still need access to Domino.class, the compiled code.

 Copyright 2006 by Ken Slonneger

New Code (Ordering Dominoes)

```
import java.util.*;

class OrderedDomino extends Domino implements Comparable
{
    OrderedDomino()  {  super();  }

    OrderedDomino(boolean b)  {  super(b);  }

    OrderedDomino(int v1, int v2, boolean b)
    {   super(v1, v2, b);  }

    public int compareTo(Object ob)
    {
        OrderedDomino d = (OrderedDomino)ob;
        int diff = getLow() - d.getLow();
        if (diff != 0)
            return diff;
        else
            return getHigh() - d.getHigh();
    }

    public boolean equals(Object ob)
    {
        if (ob instanceof OrderedDomino)
        {
            OrderedDomino d = (OrderedDomino)ob;

            return getLow()==d.getLow()
                        && getHigh()==d.getHigh();
        }
        else return false;
    }
}
```

The next class tests OrderedDomino by creating 40 random dominoes and placing them into a SortedSet.

Note that the resulting set has no duplicates and is ordered.

```
public class SortDoms
{
    public static void main(String [] args)
    {
        SortedSet sSet = new TreeSet();

        for (int m=1; m<=40; m++)
            sSet.add(new OrderedDomino(true));

        for (Iterator rat = sSet.iterator(); rat.hasNext();  )
            System.out.println(rat.next());
    }
}
```

Output

<0, 1> UP	<2, 6> UP	<4, 7> UP
<0, 2> UP	<2, 7> UP	<4, 8> UP
<0, 3> UP	<2, 9> UP	<4, 9> UP
<0, 6> UP	<3, 5> UP	<5, 5> UP
<0, 7> UP	<3, 7> UP	<5, 8> UP
<0, 9> UP	<3, 8> UP	<5, 9> UP
<1, 5> UP	<3, 9> UP	<6, 6> UP
<1, 7> UP	<4, 4> UP	<6, 7> UP
<1, 9> UP	<4, 5> UP	<7, 7> UP
<2, 2> UP	<4, 6> UP	<7, 8> UP

 Copyright 2006 by Ken Slonneger

Just Remove Duplicates

Suppose we do not need the dominoes sorted, but just want to remove the duplicates.

Change the main method in the class SortDoms to read:

```
public static void main(String [] args)
{
    Set set = new HashSet();

    for (int m=1; m<=40; m++)
        set.add(new OrderedDomino(true));

    for (Iterator rat = set.iterator(); rat.hasNext();  )
        System.out.println(rat.next());
}
```

Output

<1, 9> UP	<2, 4> UP	<6, 7> UP
<1, 5> UP	<2, 6> UP	<0, 8> UP
<2, 5> UP	<7, 9> UP	<6, 8> UP
<0, 4> UP	<1, 1> UP	<5, 6> UP
<0, 3> UP	<0, 9> UP	<2, 4> UP
<1, 9> UP	<3, 6> UP	<5, 5> UP
<6, 9> UP	<4, 6> UP	<3, 5> UP
<4, 7> UP	<1, 4> UP	<5, 5> UP
<0, 2> UP	<2, 5> UP	<0, 9> UP
<0, 2> UP	<2, 8> UP	<3, 9> UP
<4, 7> UP	<1, 8> UP	<3, 5> UP
<1, 4> UP	<1, 2> UP	<1, 3> UP
<2, 3> UP	<2, 4> UP	
<6, 8> UP	<3, 5> UP	

Oops

Although we placed the forty random dominoes into a set, duplicates still exist.

<1, 9>, <2, 5>, <4, 7>, <0, 2>, <6, 8>, <2, 4>, <0, 9>, <3, 5>, and <5, 5>

What went wrong?

The Problem

We did not override the method *hashCode* from Object, which uses the address of the object in memory as its hash value.

This means that every domino we create is considered to be different with respect to the hash table that HashSet creates.

We have violated the Basic Property of hash functions:

If ob1.equals(ob2), then ob1.hashCode() == ob2.hashCode().

Solution

Add an implementation of *hashCode* that satisfies the Basic Property to OrderedDomino.

```
public int hashCode()
{
    return 101*getLow() + 103*getHigh();
}
```

 Copyright 2006 by Ken Slonneger

Output

<4, 9> UP	<3, 8> UP	<0, 8> UP
<0, 6> UP	<4, 7> UP	<6, 9> UP
<1, 5> UP	<0, 4> UP	<3, 5> UP
<6, 7> UP	<2, 2> UP	<4, 4> UP
<3, 3> UP	<0, 9> UP	<0, 1> UP
<3, 9> UP	<2, 7> UP	<1, 6> UP
<4, 8> UP	<7, 9> UP	<6, 8> UP
<0, 5> UP	<3, 6> UP	<2, 5> UP
<5, 7> UP	<0, 2> UP	<3, 4> UP
<2, 3> UP	<1, 1> UP	

Now the duplicates are removed as the dominoes are placed into the hash table of the HashSet object.

Unique Words

Read a text file and create a list of all the words that appear in the text. No words are to be duplicated in the list.

Reading the Text File

1. Create a FileReader and use the method

   int read()

   to read the characters in the text, forming them into tokens by testing for the word delimiters.

2. Create a BufferedReader, read an entire line at a time, and use a StringTokenizer to get the words.

Both approaches were tried, reading a file containing Oliver Twist, which has over 160,000 words.

Timings

Approach 1: 157.9 seconds

Approach 2: 7.4 seconds

We use the second approach, writing a class whose objects encapsulate a Reader and respond to a *readWord* method.

 Copyright 2006 by Ken Slonneger

Java Code

```
import java.io.*;
import java.util.*;

class WordReader
{
   private BufferedReader br;
   private StringTokenizer sTok;           // null by default
   private String line;

   static final String delimiters = ".,;:-?!()\" \n\t ";

   WordReader(Reader r)
   {
      br = new BufferedReader(r);
      line = br.readLine();
      if (line == null)  line = "";
      sTok = new StringTokenizer(line, delimiters);
   }


   String readWord() throws IOException
   {                                                  // returns "" at eof
      while (!sTok.hasMoreTokens())
      {
         line = br.readLine();
         if (line == null)          // Reader at eof
            return "";
         sTok = new StringTokenizer(line, delimiters);
      }
      return sTok.nextToken();
   }
}
```

UniqueWords Class

Use a HashSet so that duplicates will be ignored.

Get the file name from the command line (args[0]).

Use the Date class to time the operation.

Put the list of words in another text file with the suffix ".out".

```
public class UniqueWords
{
   public static void main (String [] args)
   {
      Set set = new HashSet(101);        // Point

      String fname="";

      if (args.length == 1)
         fname = args[0];          // command line argument
      else
      {  System.out.println("Usage: java UniqueWords filename");
         return;
      }

      int totalWords = 0;

      try
      {
         FileReader fr = new FileReader(fname);

         WordReader wr = new WordReader(fr);

         Date d1 = new Date();

         String wd = wr.readWord();
```

```
        while (!wd.equals(""))
        {
          totalWords++;
          set.add(wd.toLowerCase());
          wd = wr.readWord();
        }
        Date d2 = new Date();
        PrintWriter pw =
          new PrintWriter(new FileWriter(fname + ".out"), true);
        printList(set, pw);
        System.out.println("Total number of words = " + totalWords);
        System.out.println("Number of unique words = " + set.size());
        System.out.println("Elapsed time = "
          + ((d2.getTime() - d1.getTime())/1000.0)  + " seconds\n");
        fr.close();
        pw.close();
      }
      catch (IOException e)
      {
        System.out.println(e);
      }
    }

    static void printList(Collection c, PrintWriter pw)
    {
      for (Iterator it = c.iterator(); it.hasNext();  )
      {
        Object word = it.next();
        pw.println(word);
      }
    }
}
```

Sample Execution

Text File: jeeves

Now, touching this business of old Jeeves--my man,
you know--how do we stand?
Lots of people think I'm much too dependent on him.
My Aunt Agatha, in fact, has even gone so
far as to call him my keeper.
Well, what I say is: Why not? The man's a genius.
From the collar upward he stands alone.
I gave up trying to run my own affairs
within a week of his coming to me.

% java UniqueWords jeeves
Total number of words = 76
Number of unique words = 65
Elapsed time = 0.014 seconds

Text File: jeeves.out

man	gone	affairs	so	dependent
a	business	lots	why	agatha
within	from	alone	is	say
he	do	we	week	on
his	how	you	think	i'm
as	own	well	much	keeper
stand	what	me	in	gave
too	collar	has	this	man's
now	to	fact	aunt	of
trying	run	people	genius	stands
not	my	upward	old	i
him	the	far	jeeves	coming
even	know	touching	call	up

Ordering the Words

Change one line (Point) in UniqueWords:

```
Set set = new TreeSet();
```

Sample Execution

% java UniqueWords jeeves
Total number of words = 76
Number of unique words = 65
Elapsed time = 0.029 seconds

Text File: jeeves.out

a	fact	in	of	to
affairs	far	is	old	too
agatha	from	jeeves	on	touching
alone	gave	keeper	own	trying
as	genius	know	people	up
aunt	gone	lots	run	upward
business	has	man	say	we
call	he	man's	so	week
collar	him	me	stand	well
coming	his	much	stands	what
dependent	how	my	the	why
do	i	not	think	within
even	i'm	now	this	you

Map Interface

A Map object will be a container holding pairs of objects
of the form (key, value) that represents a finite function.

Consequences

- the container has no duplicate keys
- each key maps to at most one value.

Example

Consider the function that maps a state name to the number
of electoral votes for that state.
Assume the numerals represent Integer objects.

{ ("Iowa", 7), ("Illinois", 21), ("Wisconsin", 10), ("Minnesota", 10),
("Michigan", 17), ("Ohio", 20), ("Missouri", 11), ("Indiana", 11),
("Pennsylvania", 21), ("Oregon", 7), ("Nebraska", 5) }

The keys in this map are the state names.

The associated values are Integer objects.

The behavior of a Map object should allow for adding and
deleting pairs and searching for keys and their values in the map.

A Map can be iterated in three ways:

By keys

By values

By the pairs, objects of type Map.Entry

 Copyright 2006 by Ken Slonneger

```
public interface Map
{
    // Basic operations
    Object get(Object key);                   // returns matching value
    Object put(Object key, Object val);       // returns previous value
    Object remove(Object key);                // returns old value
    boolean containsKey(Object key);
    boolean containsValue(Object val);
    boolean isEmpty();
    int size();

    // Bulk operations
    void clear();
    void putAll(Map m);

    // "Iterator" operations
    Set keySet();               // no duplicate keys
    Collection values();        // values may have duplicates
    Set entrySet();             // a Set of Map.Entry objects (pairs)

    public static interface Map.Entry       // a static inner interface
    {
        Object getKey();
        Object getValue();
        Object setValue(Object);
    }
}
```

SortedMap Interface

This subinterface of Map expects the keys to be kept in order according to an ordering supplied by the method *compareTo*.

The class of the keys must implement Comparable.

Additional Behavior

```
public interface SortedMap extends Map
{
    Object firstKey();
    Object lastKey();
    SortedMap subMap(Object from, Object to);
    SortedMap headMap(Object to);
    SortedMap tailMap(Object from);
    Comparator comparator();
}
```

Implementations

- Map is implemented by the class HashMap, which uses a hash table.
- SortedMap is implemented by the class TreeMap, which uses a red-black tree.

 Copyright 2006 by Ken Slonneger

HashMap Constructors

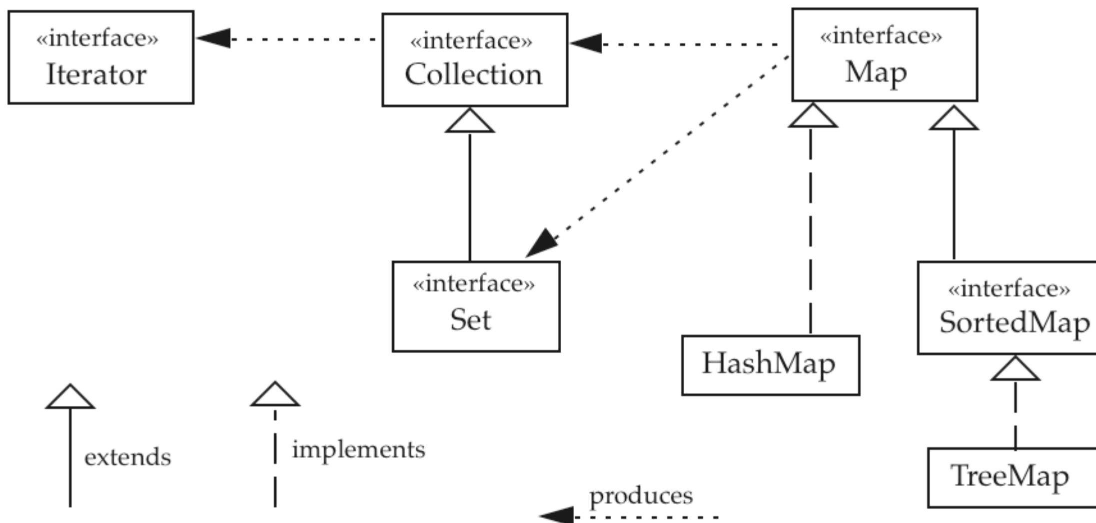
HashMap() HashMap(Map m)

HashMap(**int** sz) HashMap(**int** sz, **float** loadfac)

TreeMap Constructors

TreeMap() TreeMap(Comparator c)

TreeMap(Map m) TreeMap(SortedMap m)



Frequency Class

Alter the class UniqueWords so that it maintains a frequency count of the words found in the text file.

Use a Map that associates an Integer object with each of the words found in the text. Call the class Frequency.

Changes in *main*

```
Map table = new HashMap();     // Point
   :
String wd = wr.readWord();

while (!wd.equals(""))
{
     totalWords++;

     wd = wd.toLowerCase();

     if (!table.containsKey(wd))
          table.put(wd, new Integer(1));
     else
     {
          Integer m = (Integer)table.get(wd);

          int n = m.intValue() + 1;

          table.put(wd, new Integer(n));
     }
     wd = wr.readWord();
}
```

Changes in printList

```
static void printList(Map m, PrintWriter pw)
{
     Set keys = m.keySet();

     for (Iterator it = keys.iterator(); it.hasNext();  )
     {    Object key = it.next();
          pw.println(key + " -- " + m.get(key));
     }
}
```

 Copyright 2006 by Ken Slonneger

OR

```
Set pairs = m.entrySet();
for (Iterator it = pairs.iterator(); it.hasNext();  )
{    Map.Entry pair = (Map.Entry)it.next();
     pw.println(pair.getKey() + " -- " + pair.getValue());
}
```

Sample Execution

% **java Frequency jeeves**
Total number of words = 76
Number of unique words = 65
Elapsed time = 0.015 seconds

Text file: jeeves.out

man -- 1	how -- 1	has -- 1	old -- 1
a -- 2	own -- 1	fact -- 1	jeeves -- 1
within -- 1	what -- 1	people -- 1	call -- 1
he -- 1	collar -- 1	upward -- 1	dependent -- 1
his -- 1	to -- 3	far -- 1	agatha -- 1
as -- 1	run -- 1	touching -- 1	say -- 1
stand -- 1	my -- 4	so -- 1	on -- 1
too -- 1	the -- 2	why -- 1	i'm -- 1
now -- 1	know -- 1	is -- 1	keeper -- 1
trying -- 1	affairs -- 1	week -- 1	gave -- 1
not -- 1	lots -- 1	think -- 1	man's -- 1
him -- 2	alone -- 1	much -- 1	of -- 3
even -- 1	we -- 1	in -- 1	stands -- 1
gone -- 1	you -- 1	this -- 1	i -- 2
business -- 1	well -- 1	aunt -- 1	coming -- 1
from -- 1	me -- 1	genius -- 1	up -- 1
do -- 1			

Ordering the Words

Change one line (Point) in Frequency:

```
Map table = new TreeMap();
```

Sample Execution

% **java Frequency jeeves**

Total number of words = 76

Number of unique words = 65

Elapsed time = 0.038 seconds

Text file: jeeves.out

a -- 2	genius -- 1	man's -- 1	the -- 2
affairs -- 1	gone -- 1	me -- 1	think -- 1
agatha -- 1	has -- 1	much -- 1	this -- 1
alone -- 1	he -- 1	my -- 4	to -- 3
as -- 1	him -- 2	not -- 1	too -- 1
aunt -- 1	his -- 1	now -- 1	touching -- 1
business -- 1	how -- 1	of -- 3	trying -- 1
call -- 1	i -- 2	old -- 1	up -- 1
collar -- 1	i'm -- 1	on -- 1	upward -- 1
coming -- 1	in -- 1	own -- 1	we -- 1
dependent -- 1	is -- 1	people -- 1	week -- 1
do -- 1	jeeves -- 1	run -- 1	well -- 1
even -- 1	keeper -- 1	say -- 1	what -- 1
fact -- 1	know -- 1	so -- 1	why -- 1
far -- 1	lots -- 1	stand -- 1	within -- 1
from -- 1	man -- 1	stands -- 1	you – 1
gave -- 1			

 Copyright 2006 by Ken Slonneger

Autoboxing and Unboxing

Sometimes values of primitive types must be stored as objects. Java provides eight wrapper classes to do the job.

Primitive Type	Reference Type
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Problem: Verbose Code

- Constructors or factory methods must be employed to convert the primitive values into objects.

```
Double dob = new Double(17.2);
Integer iob = Integer.valueOf(93);
Boolean bob = new Boolean(true);
```

- Since artihmetic and testing must be done with primitive values, instance methods are needed to extract the primitive values from the objects.

```
double d = dob.doubleValue() + 5.6;
int n = 8 * iob.intValue();
if (bob.booleanValue()) System.out.println("yes");
```

An Alternative in Java 1.5

Java 1.5 provides a mechanism that performs these conversions automatically.

Autoboxing

```
Double dob = 17.2;
Integer iob = 93;
Booolean bob = true;
```

The compiler inserts the code required to place these primitive values into newly created objects.

Unboxing

```
double d = dob + 5.6;
int n = 8 * iob;
if (bob) System.out.println("yes");
```

The behavior of these two sets of code will be identical to the two sets on the previous page.

Autoboxing and Unboxing

```
dob = (iob + 87) / (dob + 309);
```

Warning: Do not let the simplicity of this code lull you. It hides several operations that will make this command much slower than if all of the variables were of primitive types.

 Copyright 2006 by Ken Slonneger

Generics in Java 1.5

Basic Principle

Errors recognized by the compiler (static errors) are much easier to debug than errors detected at runtime (dynamic errors).

Type Errors

A type error occurs when an operation or method is applied to values whose types are not what the operation or method expects.

Type Safe

A programming language is type safe if no executing program has a type error.

Strong Typing

All type errors in programs are guaranteed to be detected so that executing programs will be type safe. In addition, as much of the type checking as possible is done statically (by the compiler).

Java

Java is more strongly typed than C or C++, but its typing mechanism has a few loopholes.

Arrays: Mostly Strongly Typed

```
String [] sarray = new String [10];
Integer [] iarray = new Integer [5];
```

The compiler ensures that we put only String objects into *sarray* and Integer objects into *iarray*.

When access these arrays using subscripting, we always know what we will get.

Loophole: Upcasting

```
Object [] oarray = new Object [8];
oarray[0] = "hello";
oarray[1] = 55;          // autoboxing
oarray[2] = new Domino(2, 5, true);
oarray[3] = 84.62;       // autoboxing
```

An object of a subclass of the component class can be inserted into the array.

Collection and Map Objects

With Collection objects and Map objects we have no protection at all.

```
List list = new ArrayList();
```

Any kind of object can be added to this List object.

When we extract elements from the list, we may get any kind of object.

The compiler can do no error checking to verify the correct use of compatible types.

Errors are reported at runtime as ClassCastExceptions.

Generics

Generics, a new feature in Java 1.5, brings the same type safety of arrays to Collection and Map objects.

Idea

All of the container interfaces and classes have been defined to allow a type parameter that specifies the type of the objects allowed in the container.

Furthermore

1. The compiler ensures that only objects of the component type are allowed into the container.
2. Those objects do not need to be downcast when they are selected from the container.

Example

```
List<String> slist = new ArrayList<String>();
slist.add("one");
slist.add("two");
slist.add("three");
slist.add("four");
//   slist.add(5);        Compiler error

for (Iterator<String> it = slist.iterator(); it.hasNext(); )
    System.out.println(it.next().length());
```

Observations

1. The compiler checks that only String objects are entered into *slist*.
2. The method call *it.next()* returns an object of type String.

New for Command

Java 1.5 also provides a new version of the **for** command.

for (*ComponentType var* : *CollectionOrArrayObject*)
{
do something with the value of *var*
}

For the previous example, use:

```
for (String s : slist)
    System.out.println(s.length());
```

Restriction: The loop variable *var* is read-only.

Limitations

We can still build a container that accepts Object objects.

```
Set<Object> oset = new HashSet<Object>();
oset.add("one");
oset.add(5);
oset.add(new Domino(7, 8, true));

for (Object obj : oset)
    System.out.println(obj);
```

But combined with autoboxing, we can develop concise and reliable code to solve many problems.

```
Set<Double> dset = new HashSet<Double>();
dset.add(17.2);    dset.add(33.8);    dset.add(21.5);
dset.add(44.0);    dset.add(53.0);    dset.add(9.4);
double sum = 0.0;
for (Double d : dset) sum = sum+d;
double mean = sum/dset.size();
System.out.println("mean = " + mean);
```

Type Parameters in Methods

Generic types can be used to type parameters to methods as well as the return type of a method.

Imagine methods that take List<String>, Set<Domino>, Map<String, Integer> as parameter types.

Suppose we want to define a method that prints the elements in a generic Collection object.

How do we specify the type parameter for the Collection so that any reference type may replace it?

The solution is to use a type specification with a wild card.

Wild Cards

Three kinds of types can be specified with wild cards.

? **extends** T	Stands for any subtype of T
? **super** T	Stands for any supertype of T
?	Stands for any type

Printing a Generic Collection

```
static void printCollection(Collection<?> c, PrintWriter pw)
{
    for (Object ob : c)  pw.println(ob);
}
```

These techniques can be applied to the following programs in the lecture notes: MakeAL, MakeLL, MyLinkedList, SortDoms, UniqueWords, and Frequency.

Autoboxing and the Collection Framework

Autoboxing is particularly useful with the generic containers in the Collection framework.

The next example illustrates the use of autoboxing with a Collection object.

Problem

Create a List of ten randomly generated integers between 1 and 100, and then find the product of the ten numbers.

Since we want to store the numbers in a List, they must be each wrapped as an object.

```
import java.util.*;

public class RandInts
{
    public static void main(String [] args)
    {
        List<Integer> numList = new ArrayList<Integer>();
        for (int k=1; k<=10; k++)
            numList.add((int)(100*Math.random() + 1);
        int prod = 1;
        for (Integer n : numList)
            prod = prod*n;
        System.out.println("product = " + prod);
    }
}
```

 Copyright 2006 by Ken Slonneger

Collections Class

The Collections class in *java.util* contains a set of class methods for sorting, searching, and using Collection objects.

Comparator is an interface with an instance method that acts like a class method for comparing objects:

```
        int compare(Object ob1, Object ob2);
```

We look at a few of these methods (all are public).

```
public class Collections
{
    static void sort(List ls);
    static void sort(List ls, Comparator cmp);
    static int binarySearch(List ls, Object ob);
    static int binarySearch(List ls, Object ob, Comparator cmp);
    static void reverse(List ls);
    static void shuffle(List ls);
    static void shuffle(List ls, Random r);
    static Object max(Collection c);
    static Object max(Collection c, Comparator cmp);
    static Object min(Collection c);
    static Object min(Collection c, Comparator cmp);
    static void copy(List des, List src);
    static void fill(List ls, Object ob);
    static List nCopies(int n, Object ob);
            :
}
```

UniqueWords

The sort method can be used to put the unique words in order.

Add this code before creating the second Date object in UniqueWords:

```
List list = new ArrayList(set);
Collections.sort(list);
```

Problem

Suppose we want to sort the words so that the shortest words come first and the longest words come last.

If we could subclass String, we could override *compareTo* so that the calls to

```
s1.compareTo(s2)
```

would tell us the relative size of these two strings.

But we cannot subclass String because it is defined to be **final**.

Alternative Strategy

We define a class that implements the Comparator interface.

This strategy works because the *compare* method takes both of its arguments as regular parameters. For this method, **this** is the object that implements Comparator.

To define the *compare* method, we find the difference of the lengths of the two strings and return that value if it is nonzero. Otherwise, we simply call c*ompareTo* for the two String objects.

 Copyright 2006 by Ken Slonneger

Here is the class that implements Comparator.

```
class Order implements Comparator
{
    public int compare(Object ob1, Object ob2)
    {
        String s1 = (String)ob1;
        String s2 = (String)ob2;
        int diff = s1.length() - s2.length();
        if (diff != 0)
            return diff;
        else
            return s1.compareTo(s2);
    }
}
```

Now change the code for sorting the list to read as follows:

```
List list = new ArrayList(set);
Comparator comp = new Order();
Collections.sort(list, comp);
```

Text File: jeeves.out

a i as do he in is me my of on so to	up we far has him his how i'm man not now old own	run say the too why you aunt call even fact from gave gone	know lots much this week well what alone man's stand think agatha collar	coming genius jeeves keeper people stands trying upward within affairs business touching dependent

Frequency

The sort method can be used to put the pairs in order.

Add this code before creating the second Date object in Frequency:

```
List list = new ArrayList(table.entrySet());

Collections.sort(list, new Comparator()
        {
            public int compare(Object ob1, Object ob2)
            {
                Map.Entry me1 = (Map.Entry)ob1;
                String key1 = (String)me1.getKey();
                Map.Entry me2 = (Map.Entry)ob2;
                String key2 = (String)me2.getKey();
                return key1.compareTo(key2);
            }
        } );            // an anonymous class
```

Replace the call to printList with:

```
printList(list, pw);
```

 Copyright 2006 by Ken Slonneger

Replace the definition of printList with:

```
static void printList(List mapList, PrintWriter pw)
{
    for (Iterator it = mapList.iterator(); it.hasNext(); )
    {
        Map.Entry pair = (Map.Entry)it.next();
        pw.println(pair.getKey() + " -- " + pair.getValue());
    }
}
```

Frequency Using Collections.sort, Generics, and Autoboxing

```
import java.io.*;
import java.util.*;

public class GFrequency
{
  public static void main (String [] args)
  {
    Map<String,Integer> table =
                         new HashMap<String,Integer>();
    String fname="";
    if (args.length == 1) fname = args[0];
    else
    {  System.out.println(
             "Usage: java GFrequencey filename");
       return;
    }

    int totalWords = 0;
```

```
try
{  WordReader wr =
         new WordReader(new FileReader(fname));
   String wd = wr.readWord();
   while (!wd.equals(""))
   {
      totalWords++;
      wd = wd.toLowerCase();
      if (!table.containsKey(wd))
         table.put(wd, 1);
      else
      {
         Integer m = table.get(wd);
         table.put(wd, m+1);
      }
      wd = wr.readWord();
   }

   List<Map.Entry> list =
         new ArrayList<Map.Entry>(table.entrySet());
   Collections.sort(list,
      new Comparator<Map.Entry>()
      { public int compare(Map.Entry e1, Map.Entry e2)
         {   String key1 = (String)e1.getKey();
             String key2 = (String)e2.getKey();
             return key1.compareTo(key2);
         }
      } );
```

 Copyright 2006 by Ken Slonneger

```
        PrintWriter pw =
           new PrintWriter(new FileWriter(fname + ".out"),
                           true);

        printList(list, pw);
        pw.close();
        System.out.println("Total words = " + totalWords);
        System.out.println("Unique words = " + table.size());
     }
     catch (IOException e)
     {  System.out.println(e);  }
  }

  static void printList(List<Map.Entry> mapLt, PrintWriter pw)
  {
     for (Map.Entry pair : mapLt)
        pw.println(pair.getKey() + " -- " + pair.getValue());
  }
}
```

Another Level of Generics

The Map.Entry objects can be typed as <String,Integer> to reduce the number of downcasts in the code.

```
        List<Map.Entry<String,Integer>> list =
        new ArrayList<Map.Entry<String,Integer>>(
                                        table.entrySet());
        Collections.sort(list,
           new Comparator<Map.Entry<String,Integer>>()
           { public int compare(Map.Entry<String,Integer> e1,
                               Map.Entry<String,Integer> e2)
              {    String key1 = e1.getKey();
                   String key2 = e2.getKey();
                   return key1.compareTo(key2);
              }
           } );
        PrintWriter pw =
           new PrintWriter(new FileWriter(fname + ".out"),
                           true);
        printList(list, pw);
           :
    }

    static void printList(List<Map.Entry<String,Integer>> mapLt,
                          PrintWriter pw)
    {
       for (Map.Entry<String,Integer> pair : mapLt)
          pw.println(pair.getKey() + " -- " + pair.getValue());
    }
}
```

 Copyright 2006 by Ken Slonneger

Timings with Oliver Twist

UniqueWords

Total number of words = 163188

Number of unique words = 10981

HashSet:	7.432 seconds
TreeSet:	10.965 seconds
HashSet with Collections.sort:	9.349 seconds

Frequency

Total number of words = 163188

Number of unique words = 10981

HashMap:	10.319 seconds
TreeMap:	14.192 seconds
HashMap with Collections.sort:	11.381 seconds

Container Hierarchy