MANCHESTER 1824

The University of Manchester

COMP20012

Data Structures and Algorithms

Graham Gough

COMP20012

1: The Collections Framework

Introduction

- In this half of the course we will be looking at various data structures and algorithms.
- Continuation of earlier programming courses and David's half of COMP20012
- Mostly general, applicable to any language, but some Java specific material
- The Java Collections framework provides the background to this study
- First take another look at the framework and one extension of it

Later look at a number of topics

- Hashing and Hash tables
- Trees - data structures and algorithms
- Graphs - data structures and algorithms

The Collections Framework

- The Java Collections Framework consists a number of *interfaces*, together with a set of *classes* which implement those interfaces.
- The purpose of these interfaces is to allow collections to be manipulated in a uniform way, independently of the details of their representation.
- Since Java 1.5 the Collections framework is expressed in terms of *generics*

Java Generics

Lots of material on this on the web, the official line is at
`http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`
and
`http://java.sun.com/docs/books/tutorial/collections/`

A simple example is that of a class to construct a pair of objects. Without using Generics, this would be

```
public class Pair1
{
  private final Object first, second;
  public Pair1(Object firstElt, Object secondElt)
  {
    first = firstElt;   second = secondElt;
  }

  public Object getFirst()
  {  return first;    }

  public Object getSecond()
  {  return second;   }
```

To use this class we would do something like (although without the deliberate mistake)

```
public static void main(String [] args)
{
 Pair1 pr1 = new Pair1("February", new Integer(15));
 Pair1 pr2 = new Pair1(new Integer(15), "Feb");

 System.out.println("name length: " +
    ((String)pr1.getFirst()).length());

 System.out.println("days to end: " +
       (29 - ((Integer)pr1.getSecond()).intValue()));

 System.out.println("days to end: " +
                    (29 - ((Integer)pr1.getFirst()).intValue()));
 System.out.println("name length: " +
                         ((String)pr1.getSecond()).length());
  }
}
```

```
name length: 8
days to end: 14
Exception in thread "main" java.lang.ClassCastException:
    java.lang.String at Pair1.main(Pair1.java:29)
```

- type-casting error is *not* caught at compile-time.
- Fatal Run-Time error!
- An easy mistake to make?
- No ‘Type-Safety Net’ provided
- We ‘override’ Java’s usual type-checking mechanism

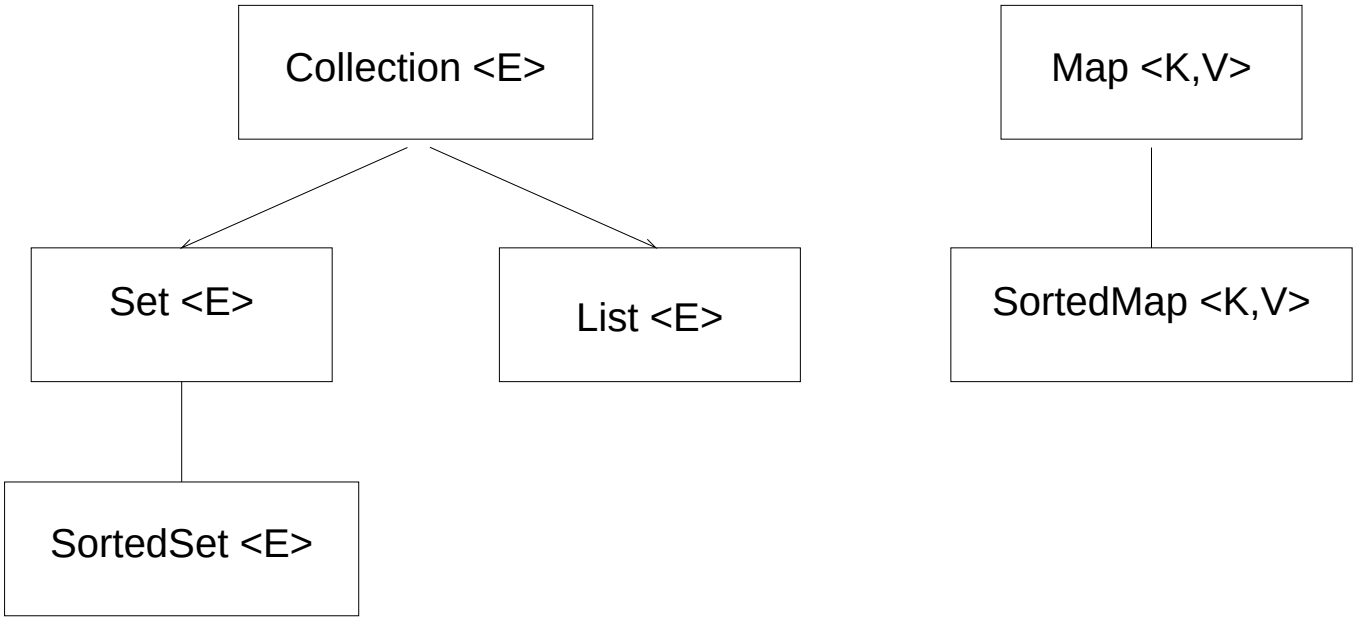
We would like this mistake to be found at compile time, not run time, enter Generics.

```
public class Pair2<S,T>
{
  private final S first;
  private final T second;

  public Pair2(S firstElt, T secondElt)
  {
    first = firstElt; second = secondElt;
  }

  public S getFirst()
  { return first;  }

  public T getSecond()
  { return second; }
```

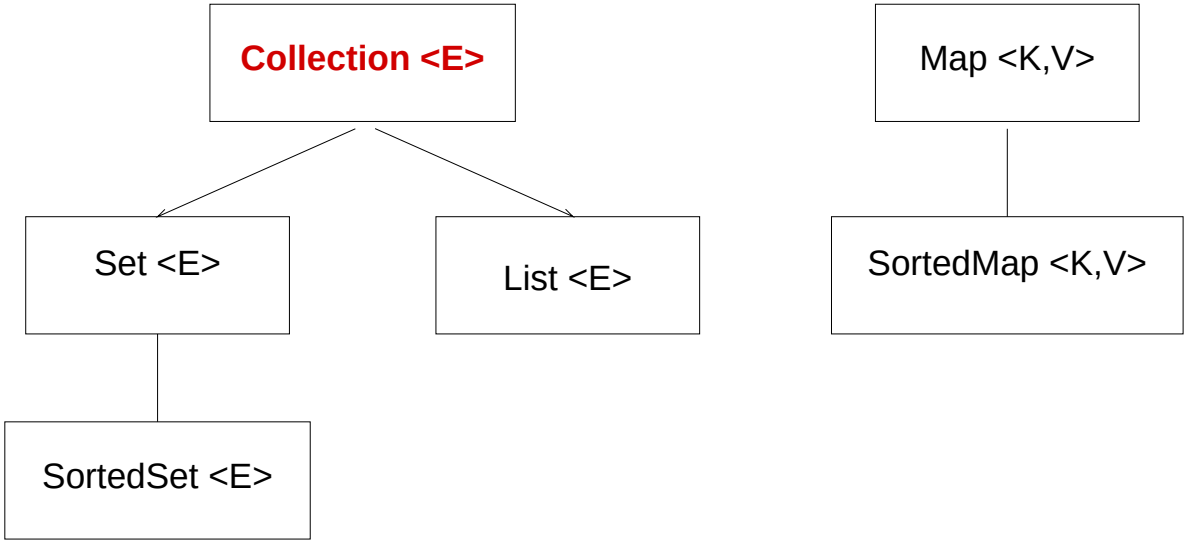
Error now detected at compile time.

```
Pair2.java:35: inconvertible types
found   : java.lang.String
required: java.lang.Integer
                    (29 - ((Integer)pr1.getFirst()).intValue()));
                                                ^
Pair2.java:37: inconvertible types
found   : java.lang.Integer
required: java.lang.String
                          ((String)pr1.getSecond()).length());
                                                ^
```

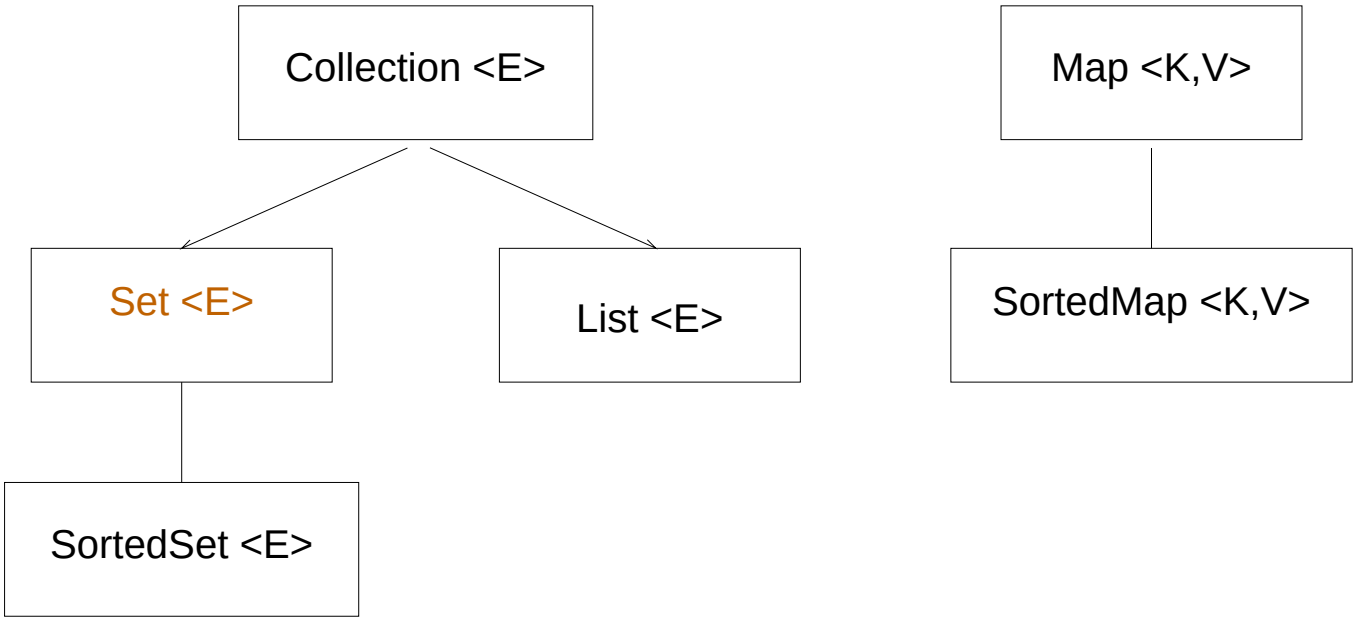
The Collections Interfaces



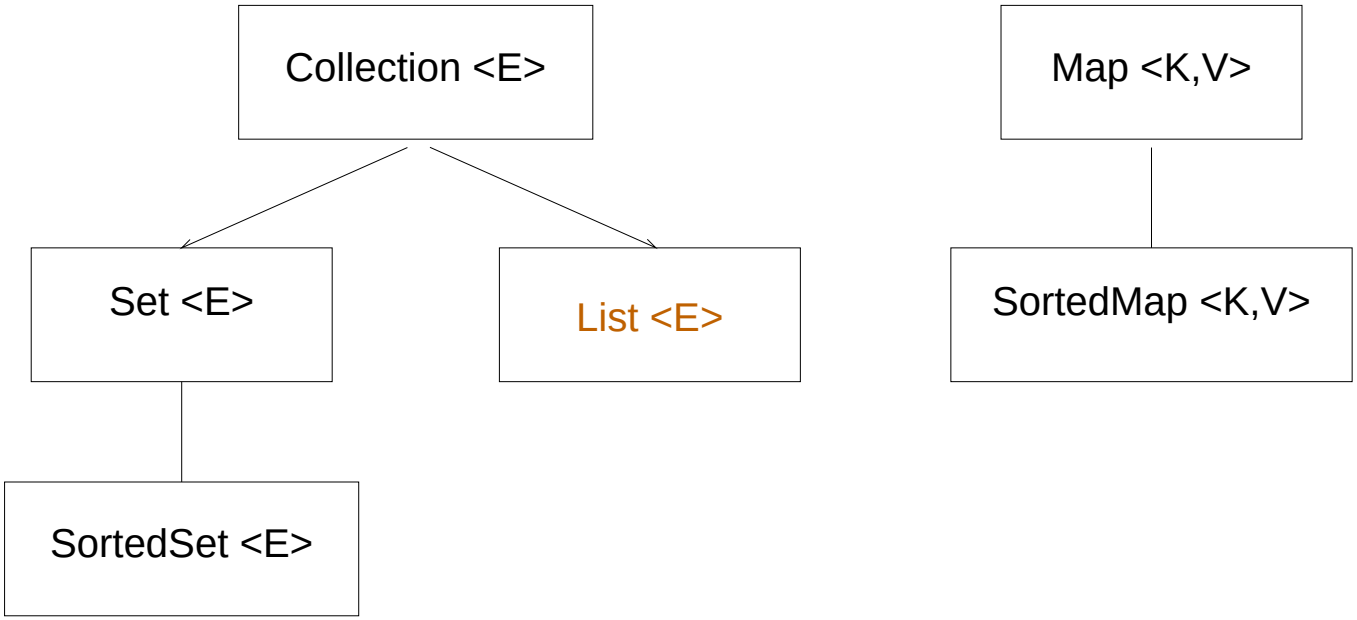
The core collection interfaces form a hierarchy.



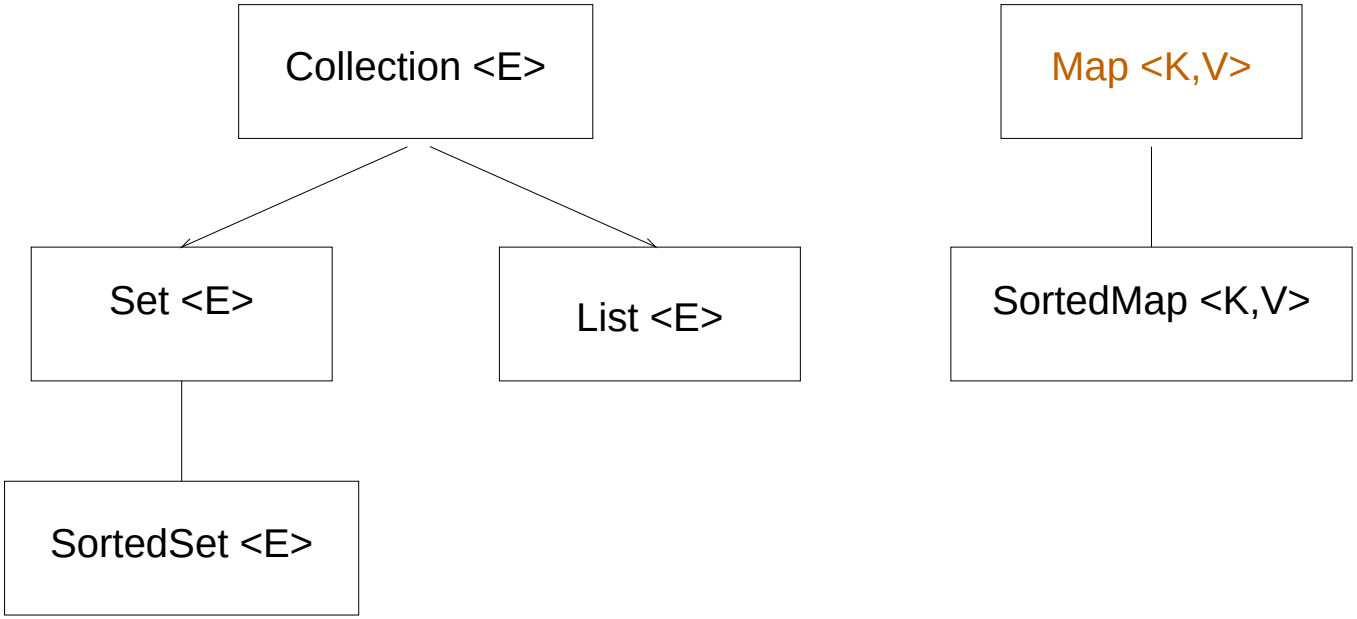
- Collection <E> - A group of objects of type E. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements.
- Least common denominator that all collections implement.
- used to pass collections around and manipulate them when maximum generality is desired.
- JDK has no direct implementations of this interface. Provides implementations of more specific subinterfaces like Set and List.



- Set <E> - The familiar mathematical set abstraction. No duplicate elements permitted. Extends the Collection interface.
- May or may not be ordered.

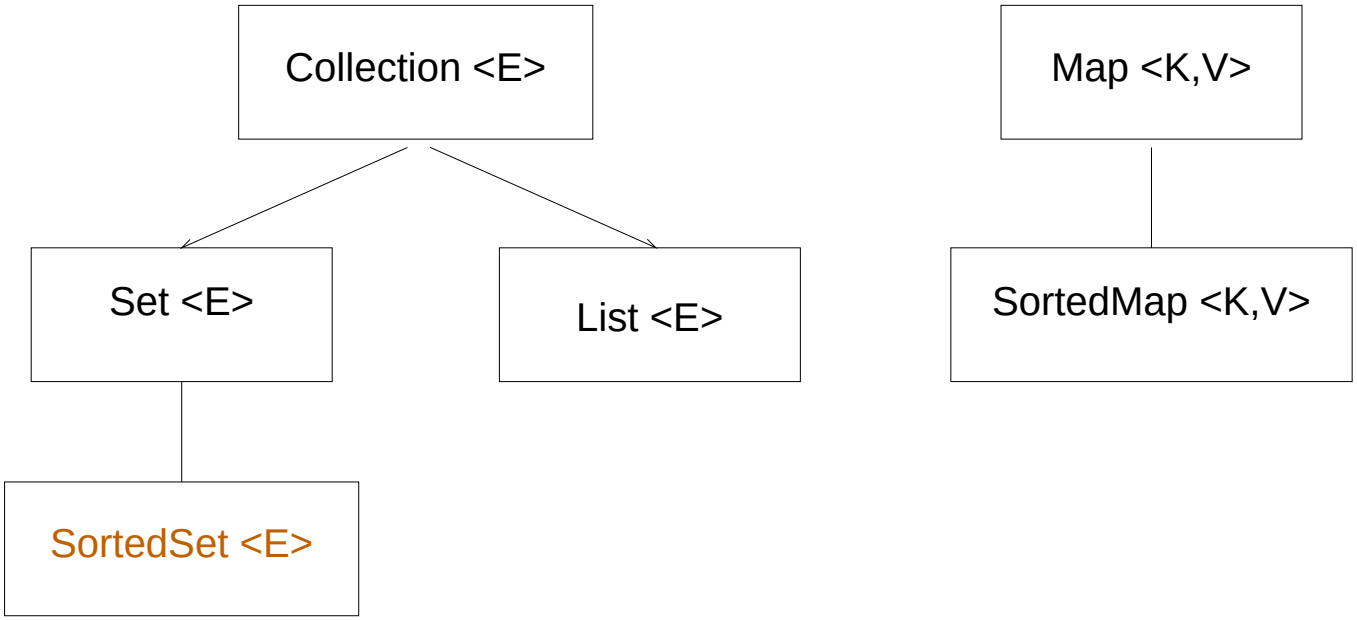


- List <E> - Ordered collection, also known as a sequence. Duplicates are generally permitted. Extends the Collection interface.
- User of a List generally has precise control over where in the List each element is inserted.
- User can access elements by their integer index (position).

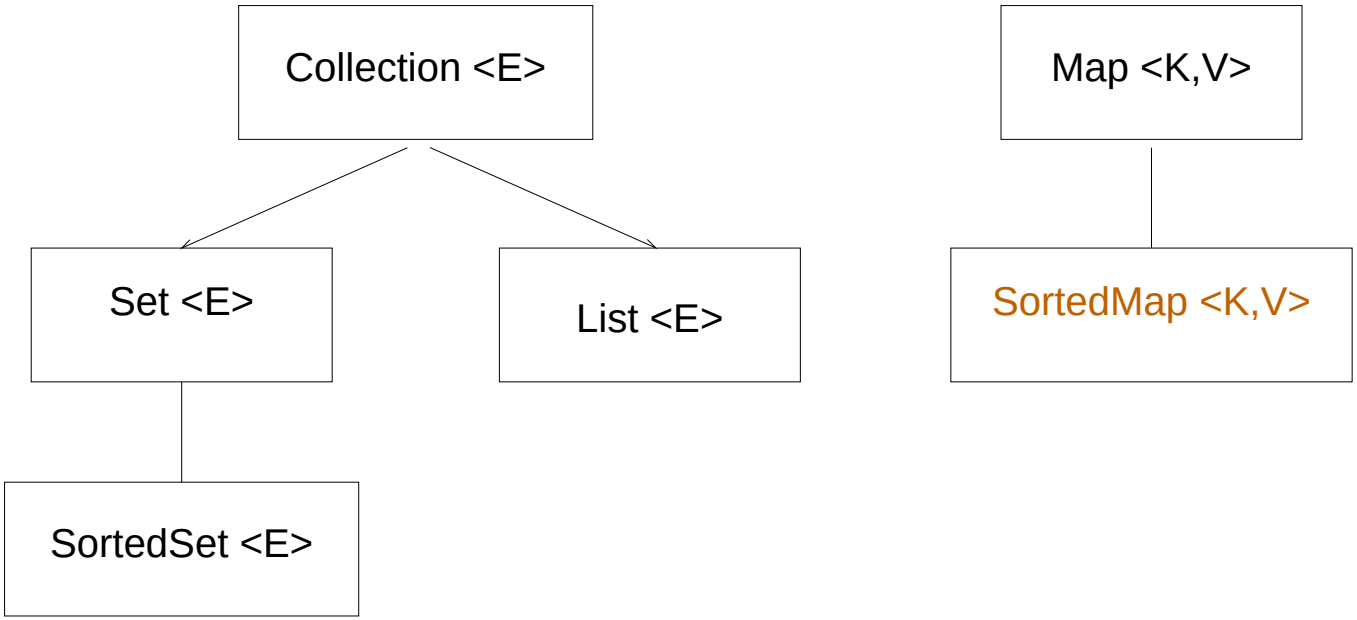


- Map <K,V> - A mapping from keys to values. Each key can map to at most one value.
- Could be a special sort of Set but is handled separately.
- The mathematical abstraction *function*.

- The last two core collection interfaces (SortedSet and SortedMap) are merely sorted versions of Set and Map.
- Objects to be stored in such a collection must be of a class which implements the `Comparable` interface
- An iterator for one of these collections will present the members of the collection in ascending element order, sorted according to the natural ordering of its elements (see Comparable), or by a Comparator provided at sorted set creation time.
- Several additional operations are provided to take advantage of the ordering.



- SortedSet - A set whose elements are automatically sorted, either in their natural ordering (see the Comparable interface), or by a Comparator object provided when a SortedSet instance is created.
- Extends the Set interface.



- SortedMap - A map whose map pairs, or maplets, are automatically sorted by key, either in the keys' natural ordering or by a comparator provided when a SortedMap instance is created.
- Extends the Map interface.
- Used for applications like dictionaries and telephone directories.

- Important to note that all restrictions on the *behaviour of methods* are precisely that
- They are not restrictions on the method of implementation

The Collection Interface

`boolean`	`add(E o)`	Ensures that this collection contains the specified element (optional operation).
`boolean`	`addAll(Collection<? extends E> c)`	Adds all of the elements in the specified collection to this collection (optional operation).
`void`	`clear()`	Removes all of the elements from this collection (optional operation).
`boolean`	`contains(Object o)`	Returns true if this collection contains the specified element.
`boolean`	`containsAll(Collection<?> c)`	Returns true if this collection contains all of the elements in the specified collection.

`boolean`	`equals(Object o)`	Compares the specified object with this collection for equality.
`int`	`hashCode()`	Returns the hash code value for this collection.
`boolean`	`isEmpty()`	Returns true if this collection contains no elements.
`Iterator <E>`	`iterator()`	Returns an iterator over the elements in this collection.
`boolean`	`remove(Object o)`	Removes a single instance of the specified element from this collection, if it is present (optional operation).
`boolean`	`removeAll(Collection <?> c)`	Removes all this collection's elements that are also contained in the specified collection (optional operation).

`boolean`	`retainAll(Collection <?> c)`	Retains only the elements in this collection that are contained in the specified collection (optional operation)
`int`	`size()`	Returns the number of elements in this collection.
`Object[]`	`toArray()`	Returns an array containing all of the elements in this collection.
`<T> T[]`	`toArray(T[] a)`	Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

The Iterator<E> interface

The Iterator interface provides a uniform mechanism for accessing all of the members of a Collection in turn

`boolean`	`hasNext()`	Returns true if the iteration has more elements.
`E`	`next()`	Returns the next element in the iteration.
`void`	`remove()`	Removes from the underlying collection the last element returned by the iterator (optional operation).

For example the following method from an earlier tutorial uses an Iterator to print out the members of an arbitrary Collection

```
 public static <T> void printCollection( Collection<T> col )
{
  Iterator<T> itr = col.iterator();
  while (itr.hasNext())
    {
      System.out.println(itr.next());
    }
}
```

or, alternatively

```
public static void printCollection( Collection<?> col )
{
  Iterator<?> itr = col.iterator();
  while (itr.hasNext())
    {
      System.out.println(itr.next());
    }
}
```

Implementations

Interface	Implementation	Historical
Set	HashSet	
	TreeSet	
List	ArrayList	Vector
	LinkedList	Stack
Map	HashMap	Hashtable
	TreeMap	Properties

We will look closer at the way some of these implementations work later in the course.

The Bag Interface

- We now define an interface that is not part of the Collections framework, together with one implementation of it.
- The mathematical concept *Bag* (or *multiset*) is that of a collection in which (as with a Set) there is no notion of order, but which differs from *Set* in that it allows the possibility of more than one copy of identical elements.
- For example the collection of coins in your pocket is, for many purposes, probably best modelled by a Bag, rather than a Set or List.
- Why is this so?

- If we did use an an alternative model for the collection of coins in your pocket what would be the implications
  - if it were a Set?
  - if it were a List?
- One way of thinking of a Bag is as a map from a Set to the natural numbers.
- How does this work?

- The Bag interface extends the Collection interface and requires a number of additional methods.
- The methods required by Collection, such as `add` and `remove`, must be implemented in a way that is consistent with the underlying mathematical model of Bag.
- So, when an item is added, the size of the Bag will *always* increase by 1, unlike the corresponding method for Set.

Additional methods

In addition to the methods specified by the `Collection` interface, we have:-

`int`	`countOccurrences(Object target)`	Find the number of occurrences of an object in a bag
`Bag`	`union(Bag b)`	Create a new bag that contains all the elements from this and `b`.
`boolean`	`removeEvery(Object togo)`	Remove *all* occurrences of an object from a bag
`boolean`	`isSubBagOf(Bag <E> other)`	Test whether a bag is a sub-bag of another. This means that every member of one must be a member of the other, taking into account the number of occurrences.

For example the bag

$$0, 1, 2, 3, 3$$

is a sub-bag of

$$0, 1, 2, 3, 3, 2$$

but not vice-versa, even though as sets they are identical.

ArrayBag: an implementation of Bag

This implementation was based on code from 'Data Structures & Other Objects using Java' by Michael Main, but extends and diverges from it in significant ways.

```
public class ArrayBag<E> implements Bag<E>
{
   //   1. The number of elements in the bag is in the instance
   //      variable numItems.
   //   2. For an empty bag, we do not care what is stored in any of
   //      data; for a non-empty bag, the elements in the bag are
   //      stored in data[0] to data[numItems-1], and we don't care
   //      what's in the rest of data.
   private static final int INITIAL_CAPACITY = 10;
   private E[ ] data;
   private int numItems;
```

ArrayBag: Constructors

This constructor allows the user to specify the initial capacity of the bag

```
/**
* Initialize an empty bag with a specified initial capacity. The
* add method works efficiently (without needing more memory)
* until this capacity is reached.
* @param initialCapacity
*   the initial capacity of this bag
* Precondition:
*   initialCapacity is non-negative.
* Postcondition:
*   This bag is empty and has the given initial capacity.
* @exception IllegalArgumentException
*   Indicates that initialCapacity is negative.
* @exception OutOfMemoryError
*   Indicates insufficient memory for:
*   new Object[initialCapacity].
**/
```

```
public ArrayBag(int initialCapacity)
{
   if (initialCapacity < 0)
      throw new IllegalArgumentException
      ("The initialCapacity is negative: " + initialCapacity);
   data = (E[]) new Object[initialCapacity];
   numItems = 0;
}

/**
* Initialize an empty bag with an initial capacity of 10.
**/

public ArrayBag( )
{
   this(INITIAL_CAPACITY);
}
```

ArrayBag: Implementing the Collections interface

add

```
/**
* Put a reference to an object into this bag.  If the addition
* would take this bag beyond its current capacity, then the
* capacity is increased before adding the new element. The new
* element may be the null reference.
* @param element
*   the element to be added to this bag
* Postcondition:
*   The element has been added to this bag, so the size of the bag
*   has increased by 1
* @exception OutOfMemoryError
*   Indicates insufficient memory for increasing the bag's capacity.
* Note:
*   An attempt to increase the capacity beyond
*   Integer.MAX_VALUE will cause the bag to fail with an
*   arithmetic overflow.
**/
```

```
public boolean add(E element)
{
   if (numItems == data.length)
   {
      // Double the capacity and add 1; ok even if numItems is 0.
      // However, in the case that numItems*2 + 1 is beyond
      // Integer.MAX_VALUE, there will be an arithmetic overflow and
      // the bag will fail.
      ensureCapacity(numItems*2 + 1);
   }

   data[numItems] = element;
   numItems++;
   return true;
}
```

addAll

```
/**
* Add references to all members of a given collection into this
* Uses add, so inherits its capabilities
* @param coll
*   the collection of elements to be added to this bag
* Postcondition:
*   All elements in coll have been added to this bag, so the size
*   of the bag has increased by the size of coll.
* @exception OutOfMemoryError
*   Indicates insufficient memory for increasing the bag's capacity.
* Note:
*   An attempt to increase the capacity beyond Integer.MAX_VALUE
*   will cause the bag to fail with an arithmetic overflow.
**/
```

```
public  boolean addAll (Collection <? extends E> coll)
{
  if (coll.isEmpty())
    return false;

  Iterator <? extends E> itr = coll.iterator();
  while (itr.hasNext())
    {
      E t = itr.next();
      add(t);
    }
  return true;
}
```

clear

```
 /**
 * Makes the bag this empty
 * @param none
 * Postcondition:
 *   this is empty
 **/
public void clear ()
{
  numItems = 0;
}
```

Note that the variable `data` is not touched.

contains

```
 /**
 * Test whether a given object is a member of this
 * @param o
 *   the object to be searched for
 * Postcondition:
 *   Returns true if and only if this bag contains o.
 **/

public boolean contains (Object o)
{
  for (int i = 0; i < numItems; i++)
    {
      if (o.equals(data[i])) {
        return true;
      }
    }
  return false;
}
```

containsAll

```
 /**
 * Test whether all members of a given collection are members of this.
 * @param coll
 *   the collection to be searched for
 * Postcondition:
 *   Returns true if and only if this bag contains all members of coll.
 **/
public boolean containsAll (Collection <?> coll)
{
  Iterator<?> itr = coll.iterator();
  while (itr.hasNext())
    {
      Object o = itr.next();
      if (!contains(o))
        return false;
    }
  return true;
}
```

equals

- For two Bags to be equal, they must contain the same (or at least `equals()`) elements in equal quantities.
- For example the Bag of coins containing
  1p, 2p, 50p, 50p, 5p, 2p, 2p
  is identical to the Bag
  2p, 1p, 50p, 2p, 50p, 5p, 2p

We implement this by using the method `isSubBagOf`

```
 /**
 * Test this and other for equality (as Bags)
 * @param other
 *   the Bag to be compared with this
 * Postcondition:
 *   Returns true if and only if this is equal (as a Bag) to other
 **/

public boolean equals (Object other)
{
  return (other instanceof Bag
              && (this.isSubBagOf((Bag)other) &&
                  ((Bag)other).isSubBagOf(this)));
}
```

hashCode

- We will say much more about hashing and hash codes later in the course.
- The requirement that the Collection interface imposes is that if two Bags are equal (as given by `equals`), the values given by `hashCode` must be equal.
- The converse is not necessary
- This is a fairly loose requirement which could be satisfied in many different ways
- One solution is simply to sum the hashCode values of all the elements in the Bag.

```
 /**
 * Return a hashcode for this
 * @param -- none
 * Postcondition:
 *   Returns an integer, and has the property if b1.equals(b2)
 *   then hashCode(b1) == hashCode(b2)
 **/

public int hashCode ()
{
  int answer = 0;
  for (int i = 0; i < numItems; i++)
    answer += data[i].hashCode();
  return answer;
}
```

isEmpty

The implementation of this method is trivial

```
/**
* Test whether this is an empty Bag
* @param -- none
* Postcondition:
*   Returns true if and only if this bag is empty.
**/

public boolean isEmpty()
{
  return (numItems == 0);
}
```

iterator

Just invoke the constructor of the appropriate iterator class (yet to be defined)

```
 /**
 * Return an Iterator for this
 * @param -- none
 * Postcondition:
 *   Returns an Iterator object for this
 **/

public Iterator<E> iterator ()
{
  return new ArrayBagIterator<E>(data,numItems);
}
```

remove

This method should remove *one* copy of a specified element from the bag.

```
/**
* Remove one copy of a specified element from this bag.
* @param target
*   an element to remove from the bag
* Postcondition:
*   If target was found in the bag, then one copy of
*   target has been removed and the method returns true.
*   Otherwise the bag remains unchanged and the method returns false.
*   Note that if target is non-null, then
*   target.equals is used to find
*   target in the bag.
**/
```

```
public boolean remove(Object target)
{
   int index; // The location of target in the data array.

   // First, set index to the location of target in the data array,
   // which could be as small as 0 or as large as numItems-1;
   // If target is not in the array, then index will be set equal
   //to numItems;
   if (target == null)
   {  // Find the first occurrence of the null reference in the bag.
      for (index = 0; (index < numItems)
             && (data[index] != null); index++)
      // No work is needed in the body of this for-loop.
      ;
   }
   else
   {  // Use target.equals to find the first occurrence of the target.
      for (index = 0; (index < numItems)
               && (!target.equals(data[index])); index++)
         // No work is needed in the body of this for-loop.
         ;
   }
```

```
    if (index == numItems)
       // The target was not found, so nothing is removed.
       return false;
    else
    {  // The target was found at data[index].
       // So reduce numItems by 1 and copy the last element
       // onto data[index].
       numItems--;
       data[index] = data[numItems];
       return true;
    }
}
```

removeAll

This method should remove *all* copies of any members of the Collection `coll` from the Bag.
Need additional method `removeEvery`, which removes *all* copies of a specified object from the Bag

```
/**
* Remove all copies of elements of a given collection from this bag
* @param coll
*   a collection of elements to be removed from the bag
* Postcondition:
*   The bag contains no members of coll, all elements of this which
*   are not members of coll remain in this.
*   If any members are removed returns true, otherwise
*   the bag remains unchanged and the method returns false.
**/
```

```
public boolean removeAll(Collection <?> coll)
{
  boolean result = false;
  Iterator <?> itr = coll.iterator();
  while (itr.hasNext())
    {
      Object o = itr.next();
      result = result || removeEvery(o);
    }
  return result;
}
```

retainAll

This method should remove *all* copies of any members of the Bag which are `not` members of the Collection `c`.

```
public boolean retainAll (Collection <?> c)
{
  boolean result = false;
  for (int i = 0; i < numItems; )
    {
      if (!c.contains(data[i])) {
        numItems--;
        data[i] = data[numItems];
        result = true;
      }
      else
        i++;
    }
  return result;
}
```

size

Return the size of the Bag

```
/**
* Determine the number of elements in this bag.
* @param - none
* @return
*   the number of elements in this bag
**/
public int size( )
{
   return numItems;
}
```

toArray

Return an array containing all of the elements in this collection.

```
public Object[] toArray ()
{
  Object[] copy = new Object[numItems];
  System.arraycopy(data, 0, copy, 0, numItems);
  return copy;
}
```

toArray

Return an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.
We're ducking out of this one

```
public <X> X[] toArray (X[] a)
{
 throw new UnsupportedOperationException("method not supported yet");
}
```

`ArrayBag`: additional methods for Bag

isSubBagOf

- One bag is a sub-bag of another if every member of one is a member of the other, taking into account the number of occurrences.
- So we just count the number of occurences of each member of this and make sure that there are at least as many occurrences in other.

```
public boolean isSubBagOf(Bag <E> other)
{
  Iterator <E> itr = this.iterator();
  while (itr.hasNext())
    {
      Object value = itr.next();
      if (this.countOccurrences(value) >
                  other.countOccurrences(value))
        return false;
    }
  return true;
}
```

countOccurrences

```
/**
* Accessor method to count the number of occurrences
* of a particular element in this bag.
* @param target
*   an element to be counted
* @return
*   The return value is the number of times that target
*   occurs in this bag. If target is non-null, then
*   the occurrences are found using the
*   target's .equals method.
**/
```

```
public int countOccurrences(Object target)
{
   int answer;
   int index;

   answer = 0;
   if (target == null)
   {  // Count how many times null appears in the bag.
      for (index = 0; index < numItems; index++)
         if (data[index] == null)
            answer++;
   }
   else
   {  // Use target.equals to determine how many times target appears.
      for (index = 0; index < numItems; index++)
         if (target.equals(data[index]))
            answer++;
   }
   return answer;
}
```

removeEvery

This method removes *all* occurrences of an object from a bag, not just one.

```
public boolean removeEvery (Object target)
{
  int index;
  boolean found = false;

  if (target == null)
    {  // Find all occurrences of the null reference in the bag.
      for (index = 0; index < numItems; )
        if (data[index] == null)
          {
            found = true;
            numItems--;
            data[index] = data[numItems];
          }
        else
          { index++; }
```

```
    }
  else
   {  // Use target.equals to find all occurrences of the target.
      for (index = 0; index < numItems; )
        if (target.equals(data[index]))
          {
            found = true;
            numItems--;
            data[index] = data[numItems];
          }
        else
          {
            index++;
          }
    }
  return found;
}
```

union

```
/**
* Create a new bag that contains all the elements from
* this and another bag.
* @param b
*   the other bag
* Precondition:
*   Neither this nor b  is null, and
*   this.getCapacity( ) + b.getCapacity( ) <= Integer.MAX_VALUE.
* @return
*   the union of this and b
* @exception NullPointerException
*   Indicates that the argument is null.
* @exception OutOfMemoryError
*   Indicates insufficient memory for the new bag.
**/
```

```
public Bag<E> union(Bag<E> b)
{
   // If b is null, then a NullPointerException thrown.
   // In the case that the total number of elements is beyond
   // Integer.MAX_VALUE, there will be an arithmetic overflow and
   // the bag will fail.
  ArrayBag<E> abthis = this;
  ArrayBag<E> abother = (ArrayBag<E>) b;
  ArrayBag<E> answer = new ArrayBag<E>(abthis.getCapacity( ) +
                                       abother.getCapacity( ));

   System.arraycopy(abthis.data, 0, answer.data,
                                       0, abthis.numItems);
   System.arraycopy(abother.data, 0, answer.data,
                        abthis.numItems, abother.numItems);
   answer.numItems = abthis.numItems + abother.numItems;

   return answer;
}
```

ArrayBag: additional methods

ensureCapacity

```
/**
* Change the current capacity of this bag.
* @param minimumCapacity
*   the new capacity for this bag
* Postcondition:
*   This bag's capacity has been changed to at least minimumCapacity.
*   If the capacity was already at or greater than minimumCapacity,
*   then the capacity is left unchanged.
* @exception OutOfMemoryError
*   Indicates insufficient memory for: new Object[minimumCapacity].
**/
```

```
private void ensureCapacity(int minimumCapacity)
{
   E biggerArray[ ];

   if (data.length < minimumCapacity)
   {
      biggerArray = (E[]) new Object[minimumCapacity];
      System.arraycopy(data, 0, biggerArray, 0, numItems);
      data = biggerArray;
   }
}
```

getCapacity

```
/**
* Accessor method to get the current capacity of this bag.
* The add method works efficiently (without needing
* more memory) until this capacity is reached.
* @param - none
* @return
*   the current capacity of this bag
**/
public int getCapacity( )
{
   return data.length;
}
```

toString

```
public String toString( )
{
  String ans = "";
  for (int i = 0 ; i < numItems; i++)
    {
      ans += data[i] + " ";
    }
  return ans;
}
```

`ArrayBagIterator`: an iterator for ArrayBag

This iterator needs to keep track of current position in the Bag.

```
public class ArrayBagIterator<T> implements Iterator<T>
{
  private int bagSize;
  // Current position
  private int current;
  private T [] contents;
```

```
/**
   * Construct the ArrayBag iterator
   * @param data the array containg the bag
   * @param size the size of the ArrayBag
   */

  ArrayBagIterator( T [] data, int size )
  {
      bagSize = size;
      current = 0;
      contents = data;
  }
```

```
/**
 * @return true if not at end of array
 */
public boolean hasNext( )
{
    return (current < bagSize);
}
```

```
/**
 * Advance the current position to the next node in the
 * If the current position is null, then do nothing.
 */
public T next( )
{
  T retval = contents[current];
  current++;
  return retval;
}
```

remove

Recall that this removes from the underlying collection the last element returned by the iterator (optional operation).

With the data structures we have used, this one is a bit tricky.
Why is this?

```
public void remove()
{
 throw new UnsupportedOperationException("method not supported yet");
}
```

An alternative approach to iterator

- Can get round the problem of lack of access to the private variables of ArrayBag by making the class which implements Iterator an *inner* class.
- It no longer needs to keep its own copy of the data and size of the Bag, since an instance of the inner class contains a reference to the object which created it.

ArrayBag2

`ArrayBag2` is identical to `ArrayBag` except for its implementation of the Iterator

```
public Iterator<T> iterator ()
{
  return new LocalArrayBagIterator();
}
```

LocalArrayBagIterator inner class

```
private class LocalArrayBagIterator implements Iterator<T>
{
  // Current position
  private int current;

  /**
   * @return true if not at end of array
   */
  public boolean hasNext( )
  {
    return (current < ArrayBag2.this.numItems);
  }
```

```
  /**
   * Advance the current position to the next node in the list.
   * If the current position is null, then do nothing.
   */
  public T next( )
  {
    return (ArrayBag2.this.data[current++]);
  }

  /**
   * Remove the object returned by the previous call to next
   */
  public void remove()
  {
    ArrayBag2.this.numItems--;
    current--;
    ArrayBag2.this.data[current] =
            ArrayBag2.this.data[ArrayBag2.this.numItems];
  }
}
```

Conclusion

- This implementation of Bag is perfectly adequate for small Bags, but will be very slow if the bags grow to any size
- What alternative implementation strategies could be used to implement Bag? Could any of the existing Collection framework classes be used?

Other new stuff in 1.5

See `http://java.sun.com/developer/technicalArticles/releases/j2se15/`

Autoboxing and Auto-Unboxing of Primitive Types

Automatic conversion between primitive types, like int, boolean, and their equivalent Object-based counterparts like Integer and Boolean.
Before

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = (list.get(0)).intValue();
```

After

```
ArrayList<Integer> list = new ArrayList<Integer>();
  list.add(0, 42);
  int total = list.get(0);
```

Enhanced for Loop

No need to use an Iterator to traverse a Collection.
Before

```
ArrayList<Integer> list = new ArrayList<Integer>();
  for (Iterator i = list.iterator(); i.hasNext();) {
        Integer value=(Integer)i.next();
    }
```

After

```
ArrayList<Integer> list = new ArrayList<Integer>();
     for (Integer i : list) { ... }
```

Enumerated Types

No need to `static final` for this

```
public enum StopLight { red, amber, green };
```

Formatted Output

`printf` rides again

Although the standard UNIX newline `'\n'` character is accepted, for cross-platform support of newlines the Java `%n` is recommended.

```
System.out.printf("name count%n");
System.out.printf("%s %5d%n", user,total);
```