Note: much of the documentation here is described in terms of interfaces, mostly to save space: in this way we don't have to describe every method for every class. Each section lists the classes we have learned about that implement the given interface. You will not need to use all the classes or methods described here.

`Collection<ElmtType>` Interface

Some classes that implement this interface are: **`ArrayList`, `LinkedList`, `TreeSet`,** and **`HashSet`.**

Selected methods:

`boolean contains(elmt)`	Returns true iff `elmt` is in this collection
`int size()`	Returns number of elements in this collection
`boolean add(elmt)`	Ensures that `elmt` is in this collection. Returns true iff this collection changed as a result of this call
`boolean remove(elmt)`	Removes an instance of `elmt` from this collection. Returns true iff this collection changed as a result of this call
`boolean isEmpty()`	Returns true iff this collection contains no elements.
`Iterator<ElmtType> iterator()`	Returns an iterator over the elements in this collection.

`Iterator<ElmtType>` Interface

Some classes that implement this interface are: **`Scanner`, `ListIterator`**

Methods:

`boolean hasNext()`

Returns `true` iff the iteration has more elements.

`ElmtType next()`

Returns the next element in the iteration. Each successive call returns a different element in the underlying collection. For `Scanner` the `ElmtType` is always `String`.

`void remove()`

Removes from the underlying collection the last element returned by the iterator. (`Scanner` does not implement this optional method.)

`Collections` Class Interface

The `Collections` class contains static methods that operate on collections. Note: `ArrayList` and `LinkedList` both implement the `List` interface used below.

Selected Methods:

`static void sort(List<ElmtType> list)`

Sorts the list into ascending order according to the natural ordering of its elements (i.e., using compareTo).

`static void sort(List<ElmtType> list, Comparator<ElementType> c)`

Sorts the list according to the order specified by the comparator.

[More other side]

`Comparator<Type>` Interface

`int compare(Type object1, Type object2)`

Must return a negative number if `object1` should come before `object2`, `0` if `object1` and `object2` are equal, or a positive number if `object1` should come after `object2`.

`Map<KeyType, ValueType>` Interface:

Two possible implementations:

`TreeMap``<KeyType, ValueType>`

`HashMap``<KeyType, ValueType>`

Selected methods:

`ValueType put(key, value)`

Associates the specified value with the specified key in this map. If the map previously contained a mapping for this key, the old value is replaced by the specified value. Returns the previous value associated with specified key, or `null` if there was no mapping for key.

`ValueType get(key)`

Returns the value to which this map maps the specified key or `null` if the map contains no mapping for this key.

`ValueType remove(key)`

Removes the mapping for this key from this map if it is present, otherwise returns `null`.

`int size()`	Number of key-value mappings in this map.
`boolean isEmpty()`	Returns true if this map contains no key-value mappings.

`Set<Map.Entry<KeyType,ValueType>> entrySet()`

Returns a set view of the entries contained in this map.

`Set<KeyType> keySet()`	Returns a set view of the keys contained in this map.

`Map.Entry<KeyType, ValueType>` Interface

`KeyType getKey()`	Return the key of the entry
`ValueType getValue()`	Return the value of the entry
`void setValue(newVal)`	Replace the current value with `newVal`

C++ `Node` type and `ListType` (this is the only part of the code handout with C++ code):

```
struct Node {
  int data;
  Node * next;
  Node() { data = 0; next = NULL; }
  Node(int d) { data = d; next = NULL; }
  Node(int d, Node * n) { data = d; next = n; }
};

typedef Node * ListType;
```

[More other side]