CMSC 202

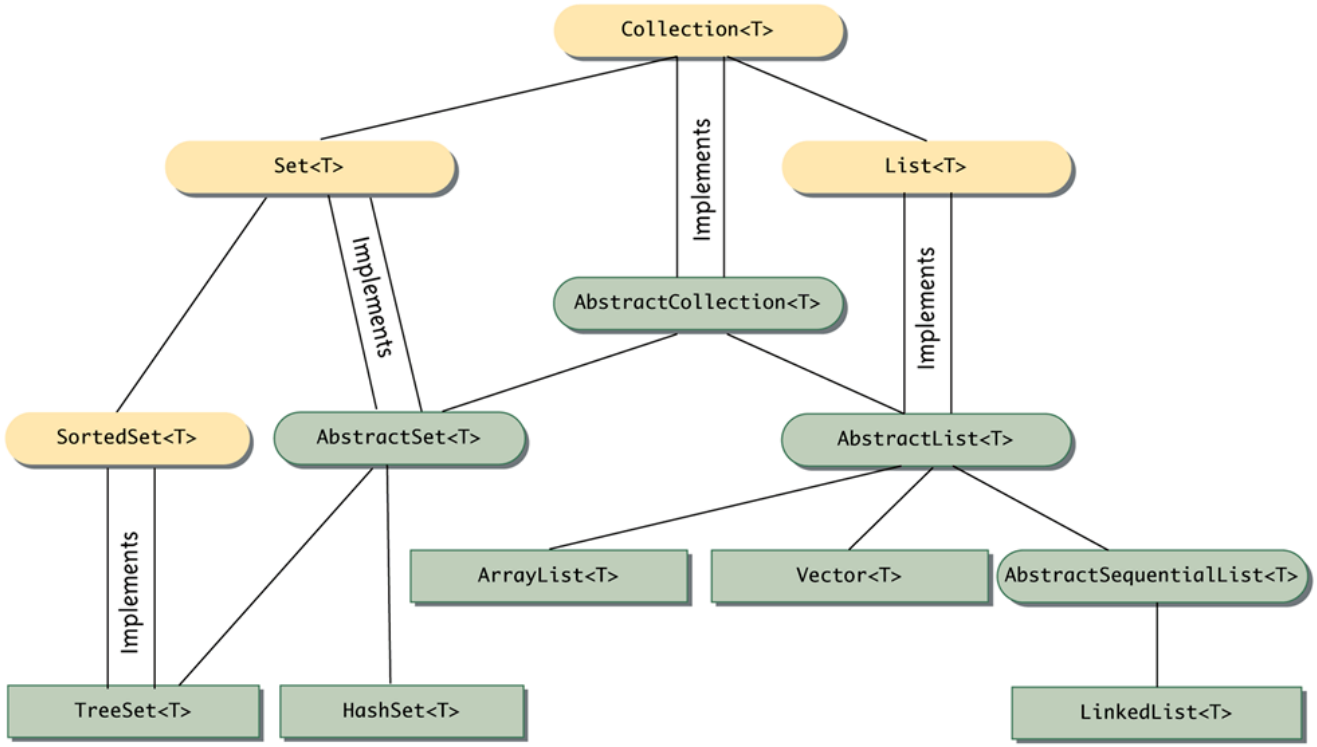
Containers

Container Definition

- A “container” is a data structure whose purpose is to hold objects.
- Most languages support several ways to hold objects.
  - Arrays are compiler-supported containers.
  - Language libraries provide a set of classes.
- Java supports two primary kinds of container interfaces
  - **Collections** contain a sequence of individual elements.
  - **Maps** contain a group of key-value object pairs.

The `Collection` Landscape

Display 16.1 **The Collection Landscape**



A single line between two boxes means the lower class or interface is derived from (`extends`) the higher one.

`T` is a type parameter for the type of the elements stored in the collection.

Wildcards

- Classes and interfaces in the collection framework can have parameter type specifications that do not fully specify the type plugged in for the type parameter.
  - Because they specify a wide range of argument types, they are known as ***wildcards***.

```
public void method(String arg1, ArrayList<?> arg2)
```

  - In the above example, the first argument is of type **`String`**, while the second argument can be an **`ArrayList<T>`** with any base type.

Wildcards

- A bound can be placed on a wildcard specifying that the type used must be an ancestor type or descendent type of some class or interface.
  - The notation **`<? extends String>`** specifies that the argument plugged in must be an object of any descendent class of **`String`**.
  - The notation **`<? super String>`** specifies that the argument plugged in must be an object of any ancestor class of **`String`**.

Collection<T> Interface

- The Collection<T> interface generalizes the concept of a *sequence* of elements.
- Basic Collection<T> operations include
  - No-argument and copy constructors
  - boolean contains(Object x) – returns true if at least one instance of x is in the collection
  - boolean containsAll(Collection<?> targets) – returns true if all targets are contained in the calling collection object
  - boolean equals(Object x) – This is equals for the collection, not the elements in the collection. Intuitive meaning.
  - Object[ ] toArray( ) – returns an array containing all of the elements
  - boolean add(T element) – ensures that the calling collection object contains the specified element. (optional)
  - boolean addAll(Collection<? extends T> collectionToAdd) – ensures that the calling collection object contains all elements of collectionToAdd (optional)
  - boolean remove(T element) – removes a single instance of the element from the calling collection object (optional)
  - boolean removeAll(Collection<?> collectionToRemove) – removes all elements contained in collectionToRemove from the calling collection object (optional)
  - void clear( ) – removes all elements from the calling collection object (optional)

Collection Relationships

- There are a number of different predefined classes that implement the `Collection<T>` interface.
  - Programmer defined classes can implement it also.
- A method written to manipulate a parameter of type `Collection<T>` will work for all of these classes, either singly or intermixed.
- There are two main interfaces that extend the `Collection<T>` interface.
  - The `Set<T>` interface
  - The `List<T>` interface

Collection Relationships

- Classes that implement the `List<T>` interface have their elements ordered as on a list.
  - Elements are indexed starting with zero.
  - A class that implements the `List<T>` interface allows elements to occur more than once.
  - The `List<T>` interface has more method headings than the `Collection<T>` interface.
  - Some of the methods inherited from the `Collection<T>` interface have different semantics in the `List<T>` interface.
  - The `ArrayList<T>` class implements the `List<T>` interface.

Some methods in the List<T> Interface

- Semantics for methods defined in Collection<T>
  - equals( ) returns true if the calling object and argument have the same element in the same order.
  - toArray( ) returns the copies of the elements (not references) in the same order.
  - add( ) places the new element at the “end” of the list.

New Methods in the `List<T>` Interface (1 of 6)

Display 16.4 Methods in the `List<T>` Interface

```
public void add(int index, T newElement) (Optional)
```

Inserts `newElement` in the calling object's list at location `index`. The old elements at location `index` and higher are moved to higher indices.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if this `add` method is not supported by the calling object.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.
Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

(continued)

New Methods in the `List<T>` Interface (Part 2 of 6)

Display 16.4 Methods in the `List<T>` Interface

```
public boolean addAll(int index,
                      Collection<? extends T> collectionToAdd) (Optional)
```

Inserts all of the elements in `collectionToAdd` to the calling object's list starting at location `index`. The old elements at location `index` and higher are moved to higher indices. The elements are added in the order they are produced by an iterator for `collectionToAdd`.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if the `addAll` method is not supported by the calling object.
Throws a `ClassCastException` if the class of one of the elements of `collectionToAdd` prevents it from being added to the calling object.
Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.
Throws an `IllegalArgumentException` if some aspect of one of the elements of `collectionToAdd` prevents it from being added to the calling object.

(continued)

New Methods in the `List<T>` Interface (Part 3 of 6)

Display 16.4 Methods in the `List<T>` Interface

```
public T get(int index)
```

Returns the object at position `index`.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index < size()
```

```
public T set(int index, T newElement) (Optional)
```

Sets the element at the specified `index` to `newElement`. The element previously at that position is returned.
Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index < size()
```

Throws an `UnsupportedOperationException` if the `set` method is not supported by the calling object.
Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.
Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.
Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

(continued)

New Methods in the `List<T>` Interface (Part 4 of 6)

Display 16.4 Methods in the `List<T>` Interface

```
public T remove(int index) (Optional)
```

Removes the element at position `index` in the calling object. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the calling object.
Throws an `UnsupportedOperationException` if the `remove` method is not supported by the calling object.
Throws an `IndexOutOfBoundsException` if `index` does not satisfy:

```
0 <= index < size()
```

(continued)

New Methods in the `List<T>` Interface (Part 5 of 6)

Display 16.4 Methods in the `List<T>` Interface

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.
Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).
Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

(continued)

New Methods in the `List<T>` Interface (Part 6 of 6)

Display 16.4 Methods in the `List<T>` Interface

```
public List<T> subList(int fromIndex, int toIndex)
```

Returns a *view* of the elements at locations `fromIndex` to `toIndex` of the calling object; the object at `fromIndex` is included; the object, if any, at `toIndex` is not included. The *view* uses references into the calling object; so, changing the view can change the calling object. The returned object will be of type `List<T>` but need not be of the same type as the calling object. Returns an empty `List<T>` if `fromIndex` equals `toIndex`.
Throws an `IndexOutOfBoundsException` if `fromIndex` and `toIndex` do not satisfy:

```
0 <= fromIndex <= toIndex <= size()
```

(continued)

Classes That Implement List<T>

- If you do not need any methods beyond those defined in List<T>, but need a List that provides fast random access to the elements (the get( ) method), use ArrayList<T> (or the older Vector<T>).
  - But note that inserting or deleting from the middle of the ArrayList or Vector will be slow.
- If you do not need fast random access, but require efficient sequential access through the list, use LinkedList<T>.
  - Inserting or deleting from the middle of the LinkedList is faster than with ArrayList or Vector.

List<T> example code

```
public class ListExample {
   public static void main( String[ ] args)
  {
        // Note the use of List<Integer> here
        List<Integer> list = new ArrayList<Integer>( );

        // add elements to the end of the list
        for (int k = 0; k < 10; k++)
              list.add(k*2);                  // autoboxing
        for (Integer k : list)
              System.out.print( k + “, “);
        System.out.println( );

        // remove element at index 4
        list.remove( 4 );
        for (Integer k : list)
              System.out.print( k + “, “);
        System.out.println( );
```

(continued)

List<T> example code

```
        // insert 99 at index 2
        list.add( 2, 99 );
        for (Integer k : list)
                System.out.print( k + “, “);
        System.out.println( );

        // change the value at index 3 to 77
        list.set( 3, 77 );
        for (Integer k : list)
                System.out.print( k + “, “);
        System.out.println( );
    }
}
// Output ---
0, 2, 4, 6, 8, 10, 12, 14, 16, 18,
0, 2, 4, 6, 10, 12, 14, 16, 18,
0, 2, 99, 4, 6, 10, 12, 14, 16, 18,
0, 2, 99, 77, 6, 10, 12, 14, 16, 18,
```

The Collections Class

In addition to the Collection<T> interface, Java provides the Collections *class* that contains static methods to operate on or return collections.

Some methods of this class related to List are:

static void reverse(List<?> list)

that reverses the contents of the specified list.

static <T extends Comparable<? super T>
void sort(List<T> list)

that sorts the specified list

static void <T>
copy(List<? super T> dest, List<? extends T> source)

copies source List to destination List

Collection Relationships

- Classes that implement the **`Set<T>`** interface do not allow an element in the class to occur more than once.
  - The **`Set<T>`** interface has the same method headings as the **`Collection<T>`** interface, but in some cases the *semantics* (intended meanings) are different.
  - Methods that are optional in the **`Collection<T>`** interface are required in the **`Set<T>`** interface.

Methods in the `Set<T>` interface

- The **`Set<T>`** interface has the same method headings as the **`Collection<T>`** interface, but in some cases the semantics are different. For example, the **`add`** methods:

`public boolean add(T element)` ***(Optional)***

If element is not already in the calling object, element is added to the calling object and true is returned. If element is in the calling object, the calling object is unchanged and false is returned.

`public boolean addAll(Collection<? extends T> collectionToAdd)`
(Optional)

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns true if the calling object changed as a result of the call; returns false otherwise. Thus, if `collectionToAdd` is `a Set<T>`, then the calling object is changed to the union of itself with `collectionToAdd`.

Classes that implement Set<T>

- All classes that implement Set<T> disallow duplicate elements.
- If you just need a collection that does not allow duplicates, use HashSet<T>.
- If you also need fast sorted element access, use TreeSet<T>.

More Collections Methods

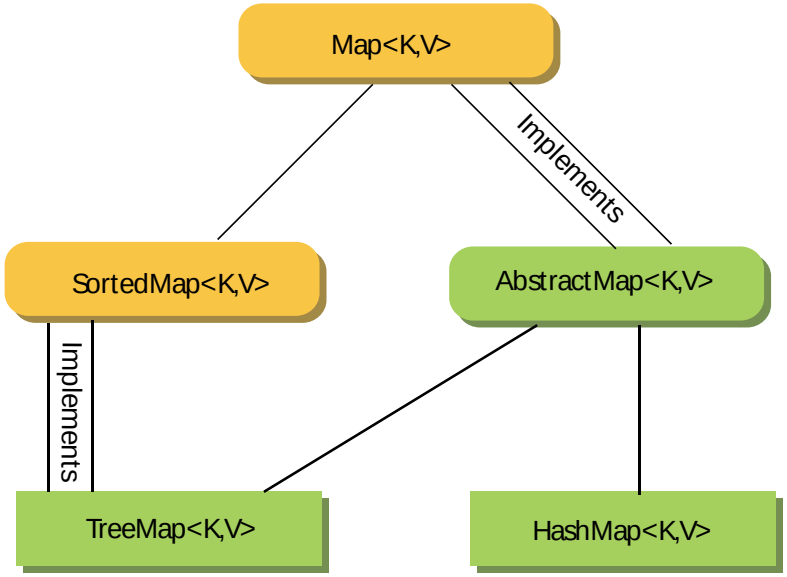
Some methods of the Collections class related to Sets include:

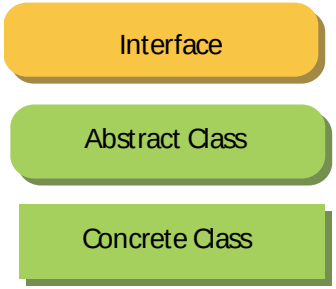
```
static <T> Set<T> singleton(T obj)
   that returns an immutable set containing only the
   specified object

static <T> Set<T> emptySet( )
   that returns an (immutable) empty set
```

The Map Framework

- The Java *map* framework deals with collections of ordered pairs.
  - For example, a key and an associated value
- Objects in the map framework can implement mathematical functions and relations, so can be used to construct database classes.
- The map framework uses the **`Map<T>`** interface, the **`AbstractMap<T>`** class, and classes derived from the **`AbstractMap<T>`** class.

The Map Landscape





A single line between two boxes means the lower class or interface is derived from (extends) the higher one.

K and V are type parameters for the type of the keys and elements stored in the map.

Basic Map<K, V> Interface

- No-argument and copy constructors
- public boolean containsValue(Object value) – returns true if the calling map object contains at least one key that maps to the specified value
- public V get(Object key) – returns the value to which the calling object maps the key. Returns null if the key is not in the map.
- public V put(K key, V value) – associates the key with the value in the map. If the key is already in the map, its old value is replaced by the value argument and returned. Otherwise null is returned. (optional)
- public void putAll(Map<? extends K, ? extends V> toAdd) – adds all mappings from toAdd to the calling map object
- public V remove(Object key) – removes the mapping for the specified key.

Classes that implement Map<K, V>

- If you require rapid access to the value associated with a key, use the HashMap<K, V> class.
  - HashMap provides no guarantee as to the order of elements placed in the map.
- If you require the elements to be in sorted order by key, then you should use the TreeMap<K,V>.
- If you require the elements to be in insertion order, use the LinkedHashMap<K,V> class.