Chapter 4

The Java Connections Framework

A *collection* is an object that is composed of individual elements.

For example, an array is a collection.

```
double[ ] salaries = new double [1000];
```

The elements in the array object (referenced by) salaries are stored contiguously, so the kTH element is at index k.

An array is a *random-access* storage structure: Any element can be accessed immediately from its index.

Drawbacks to an array:

1. Once created, an array's size is fixed.

Too big? Wasted space

Too small?

For example, if salaries (1000 double values) is too small,

```
double [ ] newSalaries = new double [2000];
```

Then

```
System.arraycopy (salaries, 0, newSalaries, 0, 1000);
```

Finally,

```
salaries = newSalaries;
```

At the end of the execution of this method, what happens to the reference newSalaries?

The array object (of 1000 double values) that salaries formerly referenced?

Drawbacks to an array:

2. **Programmer must do all the work to maintain and utilize the array.**

For examples:

To insert an element at index 30, all subsequent elements must be moved.

To print out all the elements, you must keep track of how many there are (not simply salaries.length).

Better than arrays: Instances of collection classes

A *collection class* is a class whose instances are collections.

The elements in a collection must be (references to) objects.

Primitive types (int, double, boolean, …) are not allowed, but wrapper classes can be used (Integer, Double, Boolean**). Or** String **or** FullTimeEmployee**, or … .**

A *contiguous-collection class* stores the elements in an array field.

Examples: ArrayCollection **(Lab 6),**
ArrayList **(Chapter 6),**
Heap **(Chapter 14)**

Generics: The use of type parameters in the declaration of classes and interfaces.

public class ArrayList<E>

E **(for "Element") is a type parameter.**

When an instance of the ArrayList **class is declared (and constructed), a specific type in angle brackets follows the class identifier.**

```
ArrayList <Double> salaryList = new ArrayList<Double>();
```

This creates an instance, salaryList**, of the** ArrayList **collection class. The elements in** salaryList **must be of type (reference to)** Double**.**

What if salaryList **needs to be expanded? Done automatically!**

What if you need to know the number of elements currently in salaryList?
salaryList.size()

What if you want to insert an element at index 30?

salaryList.add (30, **new** Double (40000.00));

Even easier:

salaryList.add (30, 40000.00);

This is called *boxing*: The automatic conversion of a primitive value to the appropriate wrapper object.

There is also *unboxing*: The automatic conversion of a wrapper object to the appropriate primitive value.

```
double sum = 0;
sum = sum + salaryList.get (30);
```

This statement increases sum **by the underlying double** of the **Double object at index 30 of salaryList**.

Linked-collection classes provide a widely used alternative to contiguous-collection classes.

In a *linked-collection class*, each element is stored in an Entry object that also includes at least one reference to another Entry object.

The Java collections framework consists of two hierarchies. In both of those hierarchies, there is an interface at the top, and fully defined classes at the bottom.

In between, there are *abstract classes*: Classes that may have undefined methods (like an interface) as well as defined methods (like a regular class).

What does an abstract class provide that an interface does not?

Simple definitions of methods that need not be overridden in the fully defined subclasses.

For a simple example,

```
public interface Collection<E>
{
   public int size();

   public boolean isEmpty();

   …
} // interface Collection<E>
```

E **is a type parameter.**

```
public abstract class AbstractCollection<E>
                                implements Collection<E>
{
    public abstract int size();

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

    …
} // abstract class AbstractCollection
```

The benefit is that a subclass of AbstractCollection need not override isEmpty().

The Collection interface includes method headings for inserting, removing and searching for an element in a collection.

But what if the application entails accessing all of the elements in a collection?

Print each employee whose gross pay is > $10,000.

Remove each club member who has not paid dues this year.

Determine each student's grade point average.

An *iterator* is an object that allows a user to loop through a collection without accessing the fields.

Associated with each class that implements the Collection interface, there is an iterator class that implements the following interface:

```
public interface Iterator<E>
{
    // Returns true if this Iterator object is positioned
    // at an element in the collection.
    public boolean hasNext();

    // Returns the element this Iterator object is
    // positioned at, and advances this Iterator object.
    public E next();

    // Removes the element returned by the most
    // recent call to next().
    public void remove();
} // interface Iterator
```

And, to associated an iterator object with a collection, use the following method from the Collection interface:

```
// Returns an Iterator object to iterate over this collection.
Iterator<E> iterator();
```

For example, suppose we want to print the highest salary in the ArrayList object salaryList, created earlier:

```
ArrayList<Double> salaryList =
                    new ArrayList<Double>();
```

```
Iterator<Double> itr = salaryList.iterator();

double largest = -1.00;
while (itr.hasNext())
{
        double current = itr.next();
        if (current > largest)
            largest = current;
} // while

System.out.println (“The largest salary is “ + largest);
```

In this example, and in most examples, all we want to do is access the elements: There are no calls to the remove() method.

For such situations, there is an enhanced for statement.

```
double largest = -1.00;
for (Double  current: salaryList)
    if (current > largest)
        largest = current;
System.out.println (“The largest salary is “ + largest);
```

Exercise: Replace the following with enhanced for statements. The code prints out the number of above-average salaries in salaryList. Assume that salaryList is non-empty.

```
double sum = 0.00;

Iterator<Double> itr = salaryList.iterator();
while (itr.hasNext())
    sum += itr.next();

double average = sum / salaryList.size();

itr = salaryList.iterator();
int count = 0;
while (itr.hasNext())
    if (itr.next() > average)
        count++;

System.out.println ("The number of above-average " +
                    "salaries is " + count);
```

The Collection interface has method headings for inserting, removing and searching – and a few other methods: size(), isEmpty(), toArray(),

…

The List interface extends the Collection interface by including some index-oriented methods.

```
public interface List<E> extends Collection<E>
{
   // Returns the element at position index.
   E get (int index);

   // Replaces the element at position index with
   // element, and returns the previous occupant.
   E set (int index, E element);

   // Inserts element at position index, and then
   // all elements that were at positions >= index are
   // at the next higher position.
   void add (int index, E element);

   // Removes the element at position index, returns
   // the removed element, and then all elements that were
   // at positions > index are at the next smaller position.
   E remove (int index);
```

```
   // Returns the index of the first occurrence of obj, or
   // -1 if obj is not in this List object.
   int indexOf (Object obj);

   …
} // interface List
```

The framework has two implementations of the list interface: ArrayList **and** LinkedList**.**

```
List<String> myList = new ArrayList<String>();

myList.add ("Chelebiev");
myList.add ("Culbertson");
myList.add ("Curry");
myList.add ("Dominguez");
myList.add ("Driscoll");

System.out.println (myList);
System.out.println (myList.get (2));
myList.set (3, "Amanik");
myList.add (4, "Carson");
myList.remove (5);
System.out.println ("Carson is at index " +
                    myList.indexOf ("Carson"));
for (String name: myList)
    if (name.charAt (0) == 'C')
        System.out.print (name + " ");
```

```
List<String> myList = new LinkedList<String>();

myList.add ("Chelebiev");
myList.add ("Culbertson");
myList.add ("Curry");
myList.add ("Dominguez");
myList.add ("Driscoll");

System.out.println (myList);
System.out.println (myList.get (2));
myList.set (3, "Amanik");
myList.add (4, "Carson");
myList.remove (5);
System.out.println ("Carson is at index " +
                        myList.indexOf ("Carson"));
for (String name: myList)
     if (name.charAt (0) == 'C')
         System.out.print (name + " ");
```

In general, an ArrayList **object is faster when the application frequently needs to access the elements at specific indexes.**

Why? Random-access of the underlying array

A LinkedList **object is faster when the application entails iterating through the object and often performing insertions or removals during the iteration.**

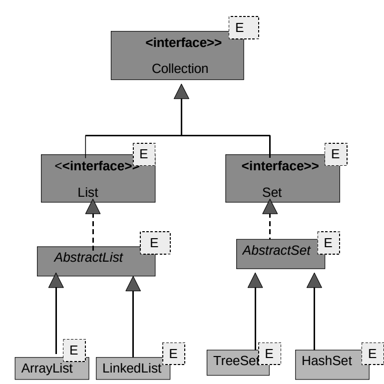
Why? At a given index, an element can be inserted or removed without moving any other elements.

The Set **interface also extends the** Collection **interface. But there are no new methods! The only change is that duplicates are not allowed in a** Set **object.**

There are two implementations of the Set **interface:**

TreeSet (Chapter 12)

HashSet (Chapter 14)



```
Set<FullTimeEmployee> employeeSet =

                    new TreeSet<FullTimeEmployee>();

employeeSet.add (new FullTimeEmployee ("Zheng 999"));
employeeSet.add (new FullTimeEmployee ("Wells 999"));
employeeSet.add (new FullTimeEmployee ("Zheng 999"));
employeeSet.add (new FullTimeEmployee ("Zheng 888"));

System.out.println (employeeSet.size());
```

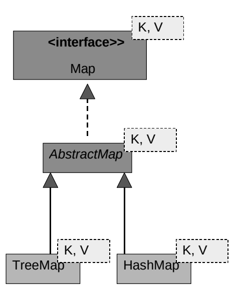
(Assume the FullTimeEmployee **class has the** equals **method based on name and gross pay.)**

When you use a TreeSet **object, elements can be inserted, in order, very quickly. Removals and searches are also very fast: worstTime(*n*) is logarithmic in n, as is averageTime(*n*).**

When you use a HashSet **object, elements can be inserted, not in order, but even quicker, on average. For inserting, removing and searching, averageTime(*n*) is constant!!**

But worstTime(*n*) is linear in n.

Finally, a *map* is a collection in which each element has two parts: A unique key and a value. The Map **interface embodies this concept.**



For example, a map of students: Each student has a unique ID (the key), and a name.

```
Map<String, String> students =
                    new TreeMap<String, String>();

students.put ("L12345678", "Stofanak");
students.put ("L01234567", "Strada");
```

The ordering is by keys, so the student with an ID of "L01234567" will come before the student with ID of "L12345678".

Exercise: Create a TreeMap

Object of taxpayers. Each key will be a social security number, and each value will be a FullTimeEmployee **object. Put two elements into the** TreeMap **object.**