Chapter 5

JAVA COLLECTIONS FRAMEWORK

A collection is an object that groups multiple objects. Collections are used to store, retrieve and manipulate data. Collections usually represent natural groupings, e.g., a Bridge hand (a collection of cards) or an address book (name to address mappings).
You should already be familiar with the idea of collections, if you've used the Java data structures such as `Vector`, `Hashtable`, or even arrays. These were different implementations of collections, but not a *framework*. The Java collections framwork is a unified architecture for representing and manipulating collections. It provides a group of interfaces and contracts which implementations must follow, as well as some sample implementations.

5.1 Generic Programming

Consider the following example:

```
public static String verb( ) {
  String[] verbs = {"eat", "eradicate" };
  return oneof( verbs );
}
```

The function of this method is to choose between the the available "verbs". This requires a trivial implementation of `oneof`—choosing a random element of the array—but requires that the words are always provided in an array.
Changing the origin of the words, such as reading them from an external source such as a file or a network requires extensive modification of the `verb` method in order to provide the correct data structure. There is an even larger problem if the list of words is too large to fit into memory. In this case providing an array becomes an impossibility.
As well as that, the modifications to `verb` are not reusable: you cannot take direct advantage of work carried out previously by others, nor can others directly reuse your work.

So the effort that went into reading the words from a file or the network may have to be repeated for the next task (such as modifying `noun`).

5.1.1 Interfaces and implementations

When the code that does the work is swamped by the implementation, it is time to separate the two. Separation means using an *interface* and providing *implementations* of that interface. Interfaces specify operations without giving any code. Implementations provide concrete code for the operations required by the interfaces.

5.1.2 Polymorphic algorithms

The last thing that is required for generic programming is polymorphic algorithms. These are algorithms which depend only on the interfaces, and not on any details of the implementations. This means that the code for these algorithms should work unchanged if the implementation changes.

5.1.3 Benefits

Improves program speed and quality Programs can be easily tuned by changing implementations to suit a particular application

Fosters software reuse New implementations which conform to existing interfaces are by their nature reusable. The same is true of new polymorphic algorithms.

5.2 The Collections Framework

The idea of the Collections Framework is to provide a generic framework for collections of data. There are several advantages that having such a framework provides in addition to the advantages of generic programming mentioned previously.

Encourage interoperability the collections interfaces become the common method of passing collections back and forth.

Reduce programming effort by providing useful data structures, programmers can concentrate on the work their code needs to do, rather than the plumbing.

Reduce effort to learn new APIs many applications naturally use collections as input and output. By providing the collections framework, such applications can make use of existing APIs, rather than providing a mini-API for dealing with their collections.

Reduce effort to design new APIs because applications can make use of collections, they can make use of the existing collections APIs.

In theory this sounds very positive, but in the past collections frameworks have been difficult to use, because there have been too many classes and interfaces to learn to be able to use the framework to the full advantage.
The Java Collections Framework has the advantages that the framework is a small set of interfaces. Sample implementations are provided as well as adapter classes to minimize the effort required to use the framework and to write new implementations of the classes, which users (including you!) are encouraged to do.

5.2.1 What is a collection?

Put simply, a collection is a group of objects. To achieve the advantages of generic programming, various operations on collections need to be provided to users. By using an example of an array, everything needed for basic collections can be observed.

```
...
Object [] a = new Object[...];
...
for (int i = 0; i < a.length; i++) {
    ... = a[i];
    a[i] = ...;
}
...
```

The code fragment has examples of:

A collection of items The array has space to hold a group of objects of a specific type.

A method of allocating space The constructor.

An iterator The for-loop variable `i` allows iteration through the elements of the collection in a well defined order.

A method of obtaining the current item By use of an assignment statement.

A method for replacing the current item An assignment statement again, which can be used for modifiable collections.

Of course, this example does not illustrate generic programming, and it is not a complete example as it does not deal with collections which can grow or shrink in size.

5.2.2 The Framework interfaces

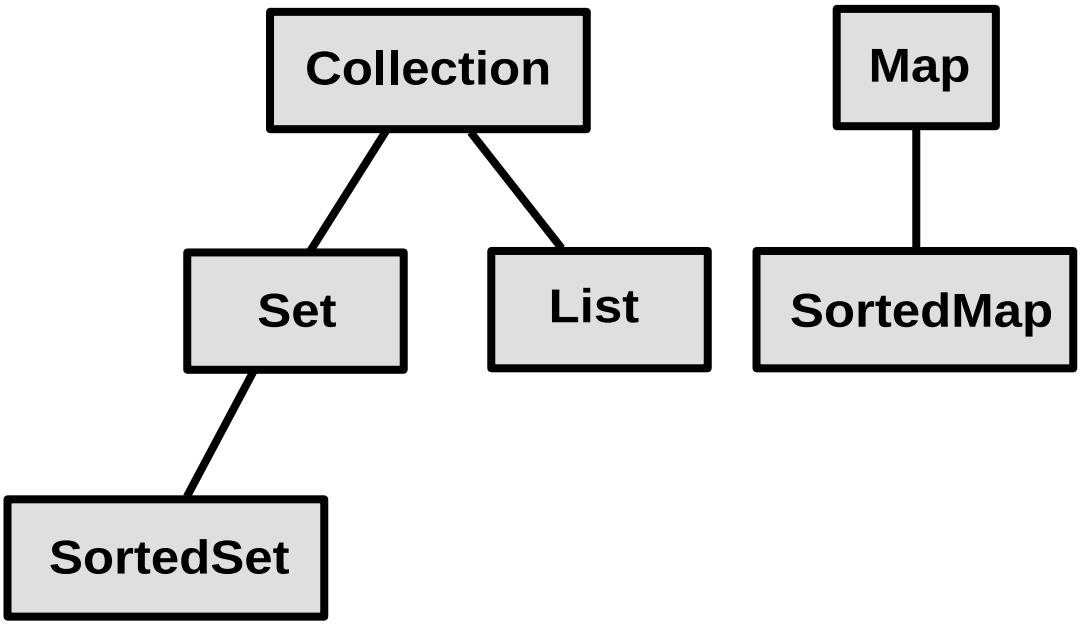
There are six interfaces, in two hierarchies.

Collection least powerful, most general. Operations include `size`, `isEmpty`, `contains`, `add`, `remove`, `iterator`.

List an ordered collection. Elements can be accessed by an integer position.

Set, SortedSet a collection that cannot contain duplicates. Same operations as collection, but is expected to prevent duplicates from occuring. Sorting determines the order which an iterator provides the elements to the user.

Map, SortedMap (*not* a collection) an object that maps keys to values; i.e., like a list, but elements can be accessed by a "key", which can be any object.



Aside In general when providing and using collections, methods which *provide* collections should provide the most specific type of collection appropriate to the application, while methods which *receive* a collection should expect the most general type of collection they are able to deal with. This increases the ability to reuse the code for different applications.

5.3 Interface Contracts

A `Collection` is more than just an interface. Implementations are expected to conform to a certain standard of behaviour. An example of the expected behaviour is af-

ter calling `c.add(o)` for a `Collection c`, `c.contains(o)` should return `true`. The expectations of behaviour form a *contract*. Users of implementations expect them to adhere to the contract. Another example of an interface contract is that for `equals` and `hashCode`. If `o1.equals(o2)`, then `o1.hashCode()` must be equal to `o2.hashCode()`.

5.4 The Interfaces in Detail

5.4.1 Collection

The `Collection` class is the most general form of a collection. There are no implementations of this class in the framework. This class is used to pass around collections when maximum generality is required. New implementations of collections which are not covered by the other classes of the framework (such as a *bag* of items which may contain duplicates) can implement this interface directly.

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

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

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Classes which implement this interface should also provide two constructors (there is no way to enforce this in Java):

- A constructor which takes no arguments;

- A constructor which takes a `Collection` as its sole argument.

The second of these two constructors is an operation to change the type of collection, for example from a `Set` to a `List`, by copying the original into the new.

5.4.2 Iterators

An `Iterator` is similar to an `Enumeration` and replace them within the context of the framework. Nonetheless there are important differences between an `Iterator` and an `Enumeration`:

- An iterator allows deletion of items from the collection with well defined semantics;
- The method names are different.

To give an idea of the differences, here is an example of enumerating the elements of a `Vector` and iterating through the elements of a `Collection`

```
...
Vector v;
...
for (Enumeration e = v.elements(); e.hasMoreElements(); ) {
    ...
    foo(e.nextElement());
    ...
}
...
```

compare the above with

```
...
Collection c;
...
for (Iterator i = c.iterator(); i.hasNext(); ) {
    ...
    foo(i.next());
    ...
}
...
```

Another important difference between `Iterator`s and `Enumerations` is the ability of `Iterator`s to *fail fast* if the underlying structure of the collection changes, for instance if a new item is added to an extensible list, while the user is iterating through the list. Under such circumstances, the iterator is permitted to throw a `ConcurrentModificationException`.

5.4.3 Implementing a `Collection`

Normally implementors of collections would subclass an `AbstractCollection`. The abstract class `AbstractCollection` provides default methods for the collection interfaces, in much the same way as the adapter classes in the Java AWT. To provide a read-only collection, subclasses must provide implementations of the `size` and `iterator` methods. For modifiable collections, subclasses must also provide `add` as well as ensuring that the `Iterator` returned by the `iterator` method implements `remove`. The default behaviour of the methods provided by the `AbstractCollection` is to throw an exception `OperationNotSupportedException`.

The operations `add` and `remove` must return true if the collection changed as a result of the operation. This is part of the contract for the `Collection`.

The two methods for providing arrays deserve mention here. The first, which takes no arguments, provides an array of the exact size required to store the collection. The second will allocate a new array if the one provided does not have sufficient space to store the entire collection, or will pad the array with `null` if the array is too large.

Keen observers will notice that the bulk operations in the class are optional to implement. This is because they can be defined in terms of the operations `add` and `remove`, which is how they are implemented in the abstract class.

5.4.4 Set

`Set`s are collections that cannot contain duplicate items. This interface is for the mathematical *set* abstraction.

Implementing a `Set` can be carried out by subclassing an `AbstractSet` in the same way that a collection can be implemented by subclassing `AbstractCollection`, with the additional constraint that the operations must adhere to the set data model, i.e., the set must contain no duplicate items.

5.4.5 List

A `List` is an indexable collection. You should already be familiar with this type of data structure from your previous Java programming experience; a `Vector` and an array are structures with similar properties. A list can contain duplicate elements, and it is possible to insert or remove anywhere from the list.

Additional operations which take advantage of this ordering are:

```
public Object get(int index)
public List subList(int fromIndex, int toIndex)
public int indexOf(Object o)
public int lastIndexOf(Object o)

// bulk operation
public boolean addAll(int index, Collection c)

// extra iterator methods
public ListIterator listIterator()
public ListIterator listIterator(int index)

// for modifiable lists
public Object set(int index, Object element)

// for resizable lists
public void add(int index, Object element)
public Object remove(int index)

protected void removeRange(int fromIndex, int toIndex)
```

which should all be reasonably self explanatory. Details can be found in the API documentation.

5.4.6 ListIterators

`ListIterator`s are extensions of `Iterator`s which allow the user to traverse lists in either direction, and allow updating of the element at the current position of the iterator. There are extra methods:

`previous`, `hasPrevious`, `nextIndex` (and `previousIndex`) for returning the index which would be returned by the appropriate call to `next` or `previous`,

`set` for updating the current element, and

`add` for adding an item to the list.

5.4.7 Adapter classes

The framework provides two abstract classes for list implementors:

AbstractList This provides default methods for implementations which are based on indexable data structures.

AbstractSequentialList This provides default methods for implementations based on sequential structures, such as linked lists.

Implementations of lists which use these adapters must provide methods for `get` and `size`, as well as `set` if the list is modifiable, and `add(int, Object)` and `remove` if the list is extendible. These adapters are different from the other abstract classes in the framework in that they do not require the user provide an `Iterator`, as it can be defined in terms of `get` and `remove`. However, users of the `AbstractSequentialList` must also provide an implementation of a `ListIterator`.

The adapter classes also provide a way of quick failure for the `Iterator` if the underlying structure changes, rather than displaying indeterminate behaviour, as happens by default.

5.4.8 Map

`Map`s are *not* `Collection`s. They are somewhat analogous to `Hashtables`, in that items in the `Map` are accessed by a *key* object which is unique to the `Map`. Each key can *map* to at most one value.

5.5 Sorting

For there to be valid implementations of `SortedSet` and `SortedMap`, there needs to be a standard method of comparing two objects for the purpose of providing a relative ordering between them. Sorting of primitive types is easy, there are predefined comparison operators (such as `<` etc.) which can be used to determine the relative ordering between two data items. However, it is not possible to have `Collections` of primitive types, we are required to use the classes `Integer` etc. For objects of non-primitive types there are two methods which can be used: *natural* ordering and *unnatural* ordering.

5.5.1 Natural ordering

Classes which implement the `Comparable` interface are said to have natural ordering. These classes must provide a method `int compareTo(Object o)`. `o1.compareTo(o2)` should return a value less than zero if `o1` is “less than” `o2`, zero if `o1` is “equal” to `o2`, and greater than zero if `o1` is “greater than” `o2`. It is normally expected that natural ordering of objects is consistent with equals. This means that `o1.equals(o2)` if and only if `o1.compareTo(o2) == 0`. One exception to this is the `BigDecimal` class in the standard classes.

When using `Comparable` objects with `SortedSet`s, there may be unexpected behaviour if the natural ordering is inconsistent with equals. This is because the contracts for `Set`s are defined in terms of `equals`, and a `SortedSet` assumes that objects which are equal according their natural ordering are also equal according to the `equals` method. There is still well defined behaviour for `SortedSet`s if the natural ordering is inconsistent with equals, although such behaviour will violate the set contract.

5.5.2 Unnatural ordering

There is a problem with natural ordering, in that the provider of the classes you are using must have implemented the `Comparable` interface. If this has not been done, there are two options:

- Create a subclass which implements this interface;
- Provide a `Comparator` object which will determine an unnatural ordering between the objects.

The first solution can get a little tedious providing wrapper classes everytime you want to create a `Collection` of a new class. It is usually simpler to implement the second alternative.

The `Comparator` class is an abstract class which allows the comparison of two arbitrary objects, although most often they are of the same type. Subclasses of `Comparator` must provide a `int compare(Object o)` which behaves in the same way as the `compareTo` method above.

The same expectations of consistency with equals apply to the `compare` method, for the reasons outlined in the previous section.

5.5.3 SortedSet and SortedMap

These are the same as the `Set` and `Map` except that the contracts guarantee that `Iterator`s return the items in sorted order. These classes will attempt to use natural order by an appropriate typecast on their members, unless a `Comparator` object is provided when constructing the collection.

There are some additional methods which take advantage of the ordering of the set or map. Those for `Set` are listed here.

```
public Comparator comparator()
public SortedSet subSet( Object fromElement,
```

```
                          Object toElement    )
public SortedSet headSet( Object toElement   )
public SortedSet tailSet( Object fromElement )
public Object first()
public Object last()
```

There are similar methods for `Map`.

The method `subSet` returns a view of the portion of this sorted set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive. If `fromElement` and `toElement` are equal, the returned sorted set is empty. Changes made to the returned set are reflected in the original.

If there are natural successors to elements in the sets, the bounds can be changed from exclusive to inclusive and vice-versa, by replacing the appropriate boundary with its successor.

The other subset methods behave in an identical manner with respect to `fromElement` and `toElement` respectively.

5.6 The `Collections` Class

As well as providing the interfaces and adapter classes, the collections framework also includes a set of static methods for manipulating collections, including methods to search lists, suffle lists, sort lists, as well as other general functions to manipulate collections.

There are also a set of factory methods for providing wrapper implementations to existing collections:

Synchronized collections by default collections are unsynchronized. These methods provide a synchronized version of the original collection. See COMPSCI 230 and/or COMPSCI 340 for explanation of synchronization.

Read-only collections These methods provide read-only views of existing collections

Finally, the `Collections` class provides a set of convenience implementations through static variables and factory methods. These are commonly used abstractions for which it is possible to provide fast implementations. There are two static variables, `EMPTY_LIST` and `EMPTY_SET`, which are immutable empty lists and sets respectively. The factory methods provide a `singletonSet`, which is an immutable set with one element, and `nCopies` which is a lightweight immutable list containing *n* copies of a single object.

5.7 General Purpose Implementations

The collection classes in Java provide some sample implementations of the interfaces.

Vector from JDK 1.0 has been retrofitted to comply with the `List` interface. There are still methods for accessing `Enumeration`s, etc, although some of the methods have changed slightly to conform to the interface.

ArrayList, LinkedList implementations of `List` using arrays and linked lists respectively. The `LinkedList` class provides extra methods which allow it to be used as a stack or queue data structure.

HashMap, HashSet implementations of `Map` and `Set` which are based on `Hashtable`s. These implementations are not sorted.

WeakHashMap like a `HashMap`, except that it allows the keys to be garbage collected while part of the map. Items accessed by garbage collected keys are removed from the mapping.

TreeMap, TreeSet a sorted `Map` or `Set` based on a balanced tree data structure which guarantees access in $\log n$ time for the elementary operations.

5.8 Writing Your Own Collection Classes

This section will be explained by way of example with an implementation of `List`, because it is the simplest. We will be adapting an array.

5.8.1 The data structure

The storage for the collection is (naturally enough) is an array. We can use the `toArray` operation to provide the "copy" constructor, and the default constructor is trivial. Note that the implementation makes use of the default methods provided by `AbstractList`.

```
public class MyArrayList extends AbstractList {
    private Object [] a;

    public MyArrayList() {
        a = new Object[0];
    }

    public MyArrayList(Collection c) {
```

```
        a = c.toArray();
    }
}
```

5.8.2 `get()` and `size()`

These are also simple, implemented by adapting the array interface:

```
    public Object get(int index) {
        return a[index];
    }

    public int size() {
        return a.length;
    }
```

5.8.3 Modifiable collections and efficiency

As an exercise, implement `set` and make the methods for `toArray` more efficient.