CT326 Programming III

LECTURE 17-19

COLLECTIONS

DR ADRIAN CLEAR
SCHOOL OF COMPUTER SCIENCE

Collections Framework

- What Is a Collection?
  - A collection (sometimes called a container) is simply an object that groups multiple elements into a single unit.
  - Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
  - Collections typically represent data items that form a natural group:
    - Like a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a collection of name-to-phone-number mappings).

Ollscoil na Gaillimhe
University of Galway

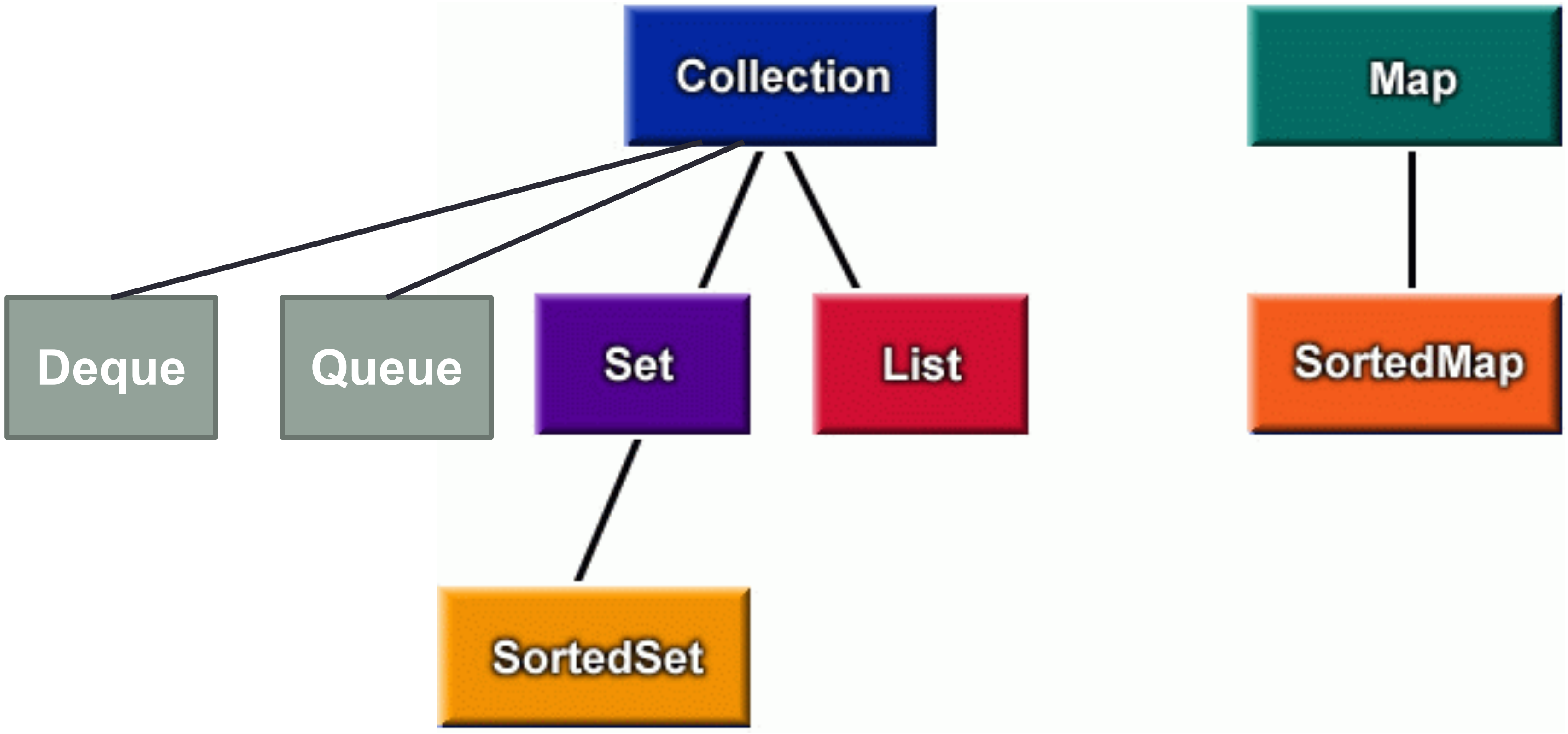
Collections Framework

- If you've used Java -- or just about any other programming language -- you're already familiar with collections.
- Collection implementations in earlier versions of Java included `Vector` , `Hashtable` , and array.
  - `Vector` and `Hashtable` are part of the `java.util` package and are widely used in existing code.
- While earlier versions of Java contained collection implementations, they did not contain a full collections framework.

Interfaces

- The core collection interfaces are the interfaces used to manipulate collections, and to pass them from one method to another.
- The basic purpose of these interfaces is to allow collections to be manipulated independently of the details of their representation.
- The core collection interfaces are the main foundation of the collections framework.

Core Collections Interfaces



Core Collections Interfaces

- The core collection interfaces form a hierarchy:
  - A `Set` is a special kind of `Collection`, and a `SortedSet` is a special kind of `Set`, and so forth.
  - Note also that the hierarchy consists of two distinct trees: a `Map` is not a true `Collection`.
- To keep the number of core collection interfaces manageable, the JDK doesn't provide separate interfaces for each variant of each collection type.
- Among the possible variants are immutable, fixed-size, and append-only.

Collections Framework

- Instead, the modification operations in each interface are designated optional:
  - A given implementation may not support some of these operations.
  - If an unsupported operation is invoked, a collection throws an `UnsupportedOperationException`.
- Implementations are responsible for documenting which of the optional operations they support.
  - All of the JDK's general purpose implementations support all of the optional operations.

`Collection` interface

- The `Collection` interface is the root of the collection hierarchy.
- A `Collection` represents a group of objects, known as its elements.
- Some `Collection` implementations allow duplicate elements and others do not.
- Some are ordered and others unordered.
- The JDK doesn't provide any direct implementations of this interface: It provides implementations of more specific sub-interfaces like Set and List.

`Collection` interface

- This interface is the lowest common denominator that all collections implement.
- `Collection` is used to pass collections around and manipulate them when maximum generality is desired.

```
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();
```

Collection interface

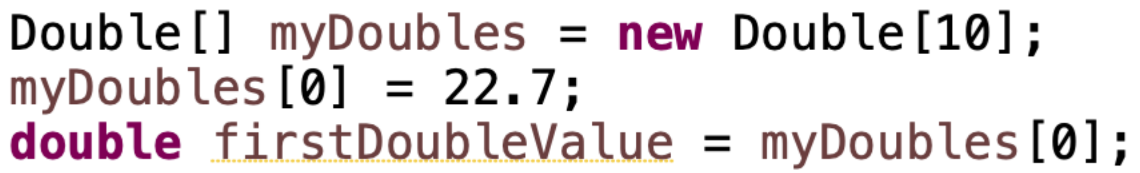
```
        // Bulk Operations
        boolean containsAll(Collection c);
        boolean addAll(Collection c);     // Optional
        boolean removeAll(Collection c); // Optional
        boolean retainAll(Collection c); // Optional
        void clear();                     // Optional

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

Type-Wrapper classes

- Collections manipulate and store *Objects* (not primitive types)
- Each Java primitive type as a corresponding **type-wrapper class** in `java.lang`
  - enable primitives to be manipulated as objects
  - Boolean, Byte, Character, Double, Float, Integer, Long, Short
  - numeric type-wrapper classes extend the `Number` class
- Methods related to primitive types are contained in the type-wrapper classes
  - e.g., `parseInt` is located in class `Integer`.

Autoboxing and Auto-unboxing

- automatic conversions between primitive types and type-wrapper objects
- A **boxing conversion** converts a value of a primitive type to an object of the corresponding type-wrapper class
- An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding primitive type
- Consider the following code:



```java
Double[] myDoubles = new Double[10];
myDoubles[0] = 22.7;
double firstDoubleValue = myDoubles[0];
```

Collections Framework

- J2SE 5.0 leaves the conversion required to transition to an Integer and back to the compiler.

- 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);
```

`Iterator` Interface

- The object returned by the `Collection iterator()` method deserves special mention.
- It is an `Iterator`, which is very similar to an `Enumeration`, but differs in two respects:
  - `Iterator` allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
  - Method names have been improved.
- The first point is important: There was no safe way to remove elements from a collection while traversing it with an `Enumeration`.

`Iterator` Interface

- The semantics of this operation were ill-defined, and differed from implementation to implementation.
- The Iterator interface is shown below:

```
public interface Iterator {
        boolean hasNext();
        Object next();
        void remove();    // Optional
    }
```

`Iterator` Interface

- The `hasNext` method is identical in function to `Enumeration.hasMoreElements`, and the `next` method is identical in function to `Enumeration.nextElement`.
- The remove method removes from the underlying `Collection` the last element that was returned by next.
  - The `remove` method may be called only once per call to next, and throws an exception if this is violated.
  - Note that `Iterator.remove` is the only safe way to modify a collection during iteration; the behaviour is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Using `Iterator`

- The following sample shows you how to use an `Iterator` to filter a `Collection`, that is, to traverse the collection, removing every element that does not satisfy some condition:

```
        static void filter(Collection<?> c) {
            for (Iterator<?> i = c.iterator(); i.hasNext(); )
                if (!cond(i.next()))
                    i.remove();
        }
    // cond() is some other related method which checks
    // the element for some condition ...
```

Using `Iterator`

- Two things should be kept in mind when looking at this simple piece of code:
  - The code is polymorphic: it works for any `Collection` that supports element removal, regardless of implementation.
  - This example shows how easy it is to write a polymorphic algorithm under the collections framework!
  - It would have been impossible to write this using `Enumeration` instead of `Iterator`, because there's no safe way to remove an element from a collection while traversing it with an `Enumeration`.

`Set` Interface

- A `Set` is a collection that cannot contain duplicate elements.
- As you might expect, this interface models the mathematical set abstraction.
- It is used to represent sets like the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- The `Set` interface has the same methods as the `Collection` interface.

`Set` Implementations

- The JDK contains two general-purpose `Set` implementations.
  - `HashSet`, which stores its elements in a hash table, is the best-performing implementation.
  - `TreeSet`, which stores its elements in a tree format, and guarantees the order of iteration.
- There follows a program that takes the words in its argument list and prints out:
  - Any duplicate words.
  - The number of distinct words.
  - A list of the words with duplicates eliminated.

Using Set

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate
                   detected: "+args[i]);

        System.out.println(s.size()+" distinct
             words detected: "+s);
    }
}
```

Using `Set`

- Running the program produces the following:

```
% java FindDups i came i saw i left
Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```

- Note that the example code always refers to the collection by its interface type (`Set`), rather than by its implementation type (`HashSet`).

Using `Set`

- Using an interface data type (like `Set`) is a strongly recommended programming practice:
  - It gives you the flexibility to change implementations merely by changing the constructor.
- If the variables used to store a collection, or the parameters used to pass it around, are declared to be of the collection's implementation type rather than its interface type:
  - Then all such variables and parameters must be changed to change the collection's implementation type.

`List` Interface

- A `List` is an ordered collection (sometimes called a sequence).
- Lists can contain duplicate elements.
- The user of a `List` generally has precise control over where in the `List` each element is inserted.
- The user can access elements by their integer index (position).
- If you've used `Vector`, you're already familiar with the general flavour of `List`.

`List` Interface

- The `List` interface is shown below:

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);
     // Optional
    void add(int index, Object element);
     // Optional
    Object remove(int index);
     // Optional
    abstract boolean addAll(int index, Collection c);
     // Optional

     // Continued on next slide ...
```

List Interface

```
    // Search
     int indexOf(Object o);
     int lastIndexOf(Object o);

     // Iteration
     ListIterator listIterator();
     ListIterator listIterator(int index);

     // Range-view
     List subList(int from, int to);
}
```

`List` Implementations

- The JDK contains two general-purpose `List` implementations.
  - `ArrayList`, which is generally the best-performing implementation.
  - `LinkedList` which offers better performance under certain circumstances.
- Also, the standard `java.util.Vector` class has been retrofitted to implement `List` in new versions of the JDK (1.2 and higher).
- The operations inherited from `Collection` all do what you'd expect them to do.

`List` Implementations

- Inserting (or removing) an element between existing elements of an `ArrayList` or `Vector` is an inefficient operation
  - for insertion, all elements at and after the index of the new one must be shifted out of the way, which could be an expensive operation in a collection with a large number of elements
  - for removal, any elements after the index of the one removed must be shifted to the left (subtracting one from their indices)
- A `LinkedList` enables efficient insertion (or removal) of elements in the middle of a collection
- A `LinkedList` is much less efficient than an `ArrayList` for jumping to a specific element in the collection

Queue Interface

- A collection for holding elements prior to processing
- Includes all basic Collection operations as well as additional insertion, removal, and inspection operations

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

`Queue` Interface

- Typically but not necessarily order elements in FIFO (first in first out) manner
- Head of queue is the element removed with remove or poll
- All new elements inserted at tail

`Deque` Interface

- A deque is a double-ended-queue
- Methods are provided to insert, remove, and examine the elements
- can be used both as last-in-first-out stacks and first-in-first-out queues
- supports the insertion and removal of elements at both end points

Deque Interface

Deque Methods

Type of Operation	First Element (Beginning of the Deque instance)	Last Element (End of the Deque instance)

Insert	`addFirst(e)`
	`offerFirst(e)`
		`addLast(e)`
		`offerLast(e)`

Remove	`removeFirst()`
	`pollFirst()`
		`removeLast()`
		`pollLast()`

Examine	`getFirst()`
	`peekFirst()`
		`getLast()`
		`peekLast()`

Map Interface

- A Map is an object that maps keys to values.
- Maps cannot contain duplicate keys:
    - Each key can map to at most one value.
    - If you've used Hashtable, you're already familiar with the general flavour of Map.
- Includes methods for basic operations (such as put, get, remove, containsKey, containsValue, size, and empty)
- Also bulk operations (such as putAll and clear)
- And collection views (such as keySet, entrySet, and values).
- The last two core collection interfaces (SortedSet and SortedMap) are merely sorted versions of Set and Map.
    - In order to understand these interfaces, you have to know how order is maintained among objects.

Map example

- Example generating frequency table of words in argument list

```
import java.util.*;
public class Freq {
   public static void main(String[] args) {
      Map<String, Integer> m = new HashMap<String, Integer>();
         // Initialize frequency table from command line
          for (String a : args) {
             Integer freq = m.get(a);
             m.put(a, (freq == null) ? 1 : freq + 1);
          }

          System.out.println(m.size() + " distinct words:");
          System.out.println(m);
   }
}
```

```
java Freq if it is to be it is up to me to delegate

8 distinct words: {to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Ollscoil na Gaillimhe
University of Galway

Object Ordering

- There are two ways to order objects:
  - The `Comparable` interface provides automatic natural order on classes that implement it.
  - While the `Comparator` interface gives the programmer complete control over object ordering.
- Note that these are not core collection interfaces, but underlying infrastructure.
  - E.g. the `Comparable` interface is implemented by classes like Byte, Float, Integer etc
- This interface imposes a total ordering on the objects of each class that implements it.

- Comparable

```
public class Person implements Comparable {
        public int compareTo(Person otherPerson) {
        …
        }
}
```

- Comparator

```
public class PersonAgeComparator implements Comparator {
        public int compare(Person aPerson, Person otherPerson)
        ...
}
```

`SortedSet` and `SortedMap` Interfaces

- A `SortedSet` is a `Set` that maintains its elements in ascending order.
  - Several additional operations are provided to take advantage of the ordering.
  - The `SortedSet` interface is used for things like word lists and membership rolls.
- A `SortedMap` is a `Map` that maintains its mappings in ascending key order.
  - It is the `Map` analogue of `SortedSet`.
  - The `SortedMap` interface is used for apps like dictionaries and telephone directories.

OLLSCOIL NA GAILLIMHE
UNIVERSITY OF GALWAY

Collection Implementation Classes

- The general-purpose JDK implementation classes are summarised in the table below:

Interface	Class	Underlying Data Structure
Set	HashSet	Hash Table
	TreeSet	Balanced Tree
List	ArrayList	Resizable Array
	LinkedList	Linked List
Map	HashMap	Hash Table
	TreeMap	BalancedTree
Queue	PriorityQueue	Heap
	LinkedList	Linked List
Deque	LinkedList	Linked List
	ArrayDeque	Resizable Array

Collections Framework

- All of the classes implement all the optional operations contained in their interfaces.
- All permit null elements, keys and values.
- Each one is unsynchronised:
  - If you need a synchronised collection, so called *synchronisation wrappers*, allow any collection to be transformed into a synchronised collection.
- All have fail-fast iterators, which detect illegal concurrent modification during iteration and fail quickly and cleanly.
  - Rather than risking arbitrary, non-deterministic behaviour at an undetermined time in the future.

Collection Algorithms

- Polymorphic algorithms are pieces of reusable functionality provided by the JDK.
  - All of them come from the `java.util.Collections` class.
- All take the form of static methods whose first argument is the collection on which the operation is to be performed.
- The great majority of the algorithms provided by the Java platform operate on `List` objects, but a couple of them (min and max) operate on arbitrary `Collection` objects.

Sorting

- The ***sort*** algorithm reorders a List so that its elements are in ascending order according to some ordering relation.
- Two forms of the operation are provided.
- The simple form just takes a List and sorts it according to its elements' natural ordering.
- The sort operation uses a slightly optimised merge sort algorithm.

Sorting

- The important things to know about the merge sort algorithm are that it is:
- *Fast*: This algorithm is guaranteed to run in n log(n) time, and runs substantially faster on nearly sorted lists.
  - Empirical studies showed it to be as fast as a highly optimised quicksort.
  - Quicksort is generally regarded to be faster than merge sort, but isn't stable, and doesn't guarantee n log(n) performance.

Sorting

- Stable: That is to say, it doesn't reorder equal elements.
    - This is important if you sort the same list repeatedly on different attributes.
    - If a user of a mail program sorts his in-box by mailing date, and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date.
    - This is only guaranteed if the second sort was stable.

Sorting

- Here's a small program that prints out its arguments in lexicographic (alphabetical) order.

```
import java.util.*;

 public class Sort {
      public static void main(String args[]) {
           List l = Arrays.asList(args);
           Collections.sort(l);
           System.out.println(l);
      }
 }
```

Sorting

- Running the program produces the following:

```
% java Sort i walk the line

[i, line, the, walk]
```

- The second form of sort takes a `Comparator` in addition to a `List` and sorts the elements with the `Comparator`.
  - Comparators can be passed to a sort method (such as `Collections.sort`) to allow precise control over the sort order.

Shuffling

- The ***shuffle*** algorithm does the opposite of what sort does: it destroys any trace of order that may have been present in a `List`.
  - That is to say, it reorders the `List`, based on input from a source of randomness, such that all possible permutations occur with equal likelihood (assuming a fair source of randomness).
- This algorithm is useful in implementing games of chance.
  - For example, it could be used to shuffle a `List` of `Card` objects representing a deck.

Shuffling

- Shuffle can also be useful for generating test cases.
- There are two forms of this operation.
- The first just takes a `List` and uses a default source of randomness.
- The second requires the caller to provide a `Random` object to use as a source of randomness.

Routine Data Manipulation

- The `Collections` class provides three algorithms for doing routine data manipulation on `List` objects.
- All of these algorithms are pretty straightforward:
- ***reverse***: Reverses the order of the elements in a List.
- ***fill***: Overwrites every element in a List with the specified value.
  - This operation is useful for re-initialising a List.

Routine Data Manipulation

- ***copy***: Takes two arguments, a destination `List` and a source `List`, and copies the elements of the source into the destination, overwriting its contents.
  - The destination `List` must be at least as long as the source.
  - If it is longer, the remaining elements in the destination `List` are unaffected.

Searching

- The ***binarySearch*** algorithm searches for a specified element in a sorted List using the binary search algorithm.
- There are two forms of this algorithm.
- The first takes a `List` and an element to search for (the "search key").
- This form assumes that the `List` is sorted into ascending order according to the natural ordering of its elements.

Searching

- The second form of the call takes a `Comparator` in addition to the `List` and the search key, and assumes that the `List` is sorted into ascending order according to the specified `Comparator`.
- The sort algorithm (described already) can be used to sort the `List` prior to calling `binarySearch`.
- The return value is the same for both forms:
  - If the `List` contains the search key, its index is returned.
  - Otherwise, the return value is (-(insertion point) - 1).

Searching

- In the negative case, the insertion point is defined as the point at which the value would be inserted into the `List`:
  - The index of the first element greater than the value, or `list.size()` if all elements in the `List` are less than the specified value.
  - This admittedly ugly formula was chosen to guarantee that the return value will be >= 0 if and only if the search key is found.
  - It's basically a hack to combine a boolean ("found") and an integer ("index") into a single int return value.

Searching

- The following idiom, usable with both forms of the `binarySearch` operation, looks for the specified search key, and inserts it at the appropriate position if it's not already present:

```
int pos = Collections.binarySearch(l, key);
        if (pos < 0)
            l.add(-pos-1, key);
```

Finding Extreme Values

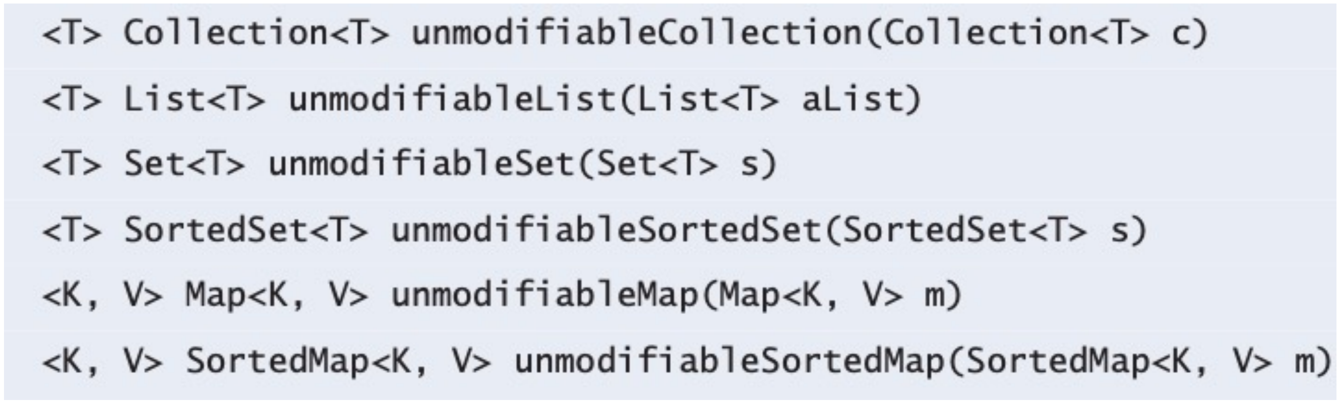
- The ***min*** and ***max*** algorithms return, respectively, the minimum and maximum element contained in a specified Collection.
- Both of these operations come in two forms.
  - The simple form takes only a Collection, and returns the minimum (or maximum) element according to the elements' natural ordering.
  - The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.

Finding Extreme Values

- These are the only algorithms provided by the Java platform that work on arbitrary Collection objects, as opposed to List objects.
- Like the fill algorithm, described earlier, these algorithms are quite straightforward to implement.
- They are included in the Java platform solely as a convenience to programmers.
- Most programmers will probably never need to implement their own collections classes, but this can be done if necessary.

Unmodifiable collections

- Collections class provides a set of static methods that create *unmodifiable wrappers* for collections
  - throw `UnsupportedOperationException`s if attempts are made to modify the collection
- references stored in the collection are not modifiable, but the objects they refer are modifiable (unless they belong to an immutable class like `String`)



```
<T> Collection<T> unmodifiableCollection(Collection<T> c)
<T> List<T> unmodifiableList(List<T> aList)
<T> Set<T> unmodifiableSet(Set<T> s)
<T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)
<K, V> Map<K, V> unmodifiableMap(Map<K, V> m)
<K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, V> m)
```

Generic Types

- The Collections API provides common functionality like LinkedLists, ArrayLists and HashMaps that can be used by more than one Java type.
- The next example uses the 1.4.2 libraries and the default javac compile mode:

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

Collections Framework

- The cast to Integer on the last line is an example of the typecasting issues that generic types aim to prevent.
- The issue is that the 1.4.2 Collection API uses the Object class to store the Collection objects, which means that it cannot pick up type mismatches at compile time.
- The same example with the generified Collections library is written as follows:

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

Collections Framework

- The user of a generified API has to simply declare the type used at compile type using the <> notation.
- No casts are needed and in this example trying to add a `String` object to an Integer typed collection would be caught at compile time.
- Generic types therefore enable an API designer to provide common functionality that can be used with multiple data types and which also can be checked for type safety at compile time.
- Designing your own Generic APIs is a little more complex than simply using them. To get started look at the `java.util.Collection` source and also the API guide.

Next time…

- Graphical User Interfaces