## Lecture 30

To reinforce our understanding of generic data structures and of Java interfaces, we explore a part of the Java class library, the Java 2 Collections Framework.

We look at various Java interfaces (and their inheritance relationships), and various concrete classes that implement these interfaces.

Armed with this material and material from CS2010, when next we write a program that needs to maintain a collection of data, maybe we won't unthinkingly use arrays. Maybe we'll make a more considered choice.

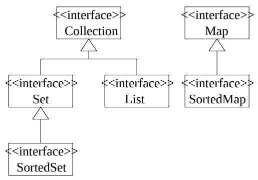
We'll use arrays if they are appropriate (in the light of their strengths/weaknesses).

But, where they're not appropriate, we'll decide on an appropriate Java interface and pick an implementation of that interface whose performance properties suit our task.



## Collections

- A *collection* is an object that contains other objects
- Some of the Java interfaces in Java 2's Collections Framework (all in `java.util`):




- Note that there can be inheritance between interfaces as well as classes (even multiple inheritance)



## The `Collection` interface

The Java interface `java.util.Collection` includes the following:

| | |
|---|---|
| `add(o)` | Add object `o` to the collection |
| `contains(o)` | Returns `true` if this collection contains an element that equals `o` |
| `isEmpty()` | Returns `true` if this collection contains no elements |
| `iterator()` | Returns an iterator over the elements in this collection |
| `remove(o)` | Removes an element that equals `o` from this collection if such an element exists |
| `size()` | Returns the number of elements in this collection |



## Sets

- A *set* is a collection of objects in which there are no duplicates and order is not significant
- Java's `Set` interface inherits from `Collection` but changes the semantics of certain methods, e.g.:

| | |
|---|---|
| `add(o)` | Add object `o` to the set if it is not already present |



## Lists

- A *list* (or *sequence*) is a collection in which duplicates are allowed & order is significant. Elements are indexed starting from 0
- Java's `List` interface inherits from `Collection` but changes the semantics of certain methods and declares additional methods, e.g.:

| | |
|---|---|
| `add(i, o)` | Inserts object `o` at position `i` |
| `add(o)` | Appends object `o` to end of list |
| `get(i)` | Returns element at position `i` |
| `indexOf(o)` | Returns the index of the first occurrence of `o` |
| `remove(i)` | Removes & returns the element at the `i`th position |
| `remove(o)` | Removes the first occurrence of object `o` |
| `set(i, o)` | Replaces the element at the `i`th position by `o` and returns the replaced element |



## Maps

- A *map* (or *function* or *dictionary* or *associative array*) is a collection of key-value pairs. Duplicate keys are not allowed & order is not significant
- Java's `Map` interface includes the following (plus `isEmpty()` and `size()`):

| | |
|---|---|
| `containsKey(k)` | Returns `true` if this map contains a mapping for key `k` |
| `containsValue(v)` | Returns `true` if there is a mapping to value `v` |
| `get(k)` | Returns the value to which key `k` is mapped |
| `keySet()` | Returns a `Set` containing all the keys in this map |
| `put(k, v)` | Adds a mapping from `k` to `v` |
| `remove(k)` | Removes the mapping for key `k` from this map |
| `values()` | Returns a `Collection` of all the values in this map |

### Which Java interface would you use?

- The staff who have been with a company for over 10 years
- The entries in a glossary in a technical manual
- The cars which are owned by a car hire company
- The destinations of lorries owned by a haulage company
- The computers which make up a computer network
- The messages that a computer in a network receives



### Implementations

Once you've decided your collection's interface, you choose an implementation (or implement a new one):

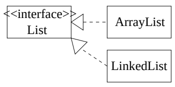
| Interface | Implementations | |
|---|---|---|
| `Set` | `HashSet` & its subclasses | A hash table |
| `List` | `ArrayList` | Resizable-array |
| | `Vector` | Resizable-array (from Java 1) |
| | `LinkedList` | Doubly linked list |
| `Map` | `HashMap` & its subclasses | Hash table |
| | `Hashtable` | Hash table (from Java 1) |
| `SortedSet` | `TreeSet` | Balanced binary tree |
| `SortedMap` | `TreeMap` | Balanced binary tree |

In all the implementations, where Java needs to test for equality of objects in the collection, Java uses the `equals` method



### Implementations, cont'd

- Showing diagrammatically that a class implements an interface:




- Sorted sets/maps are like sets/maps but the elements are sorted
- Java will have to know how to compare the objects
- One way is for the objects to implement the `Comparable` interface and include a `compareTo` method



### Iteration

- Client code often needs to visit each object in a collection
- But the code to walk through, e.g., a linked list and a resizable-array are different. Iterators provide a *uniform* way of writing this code
- If `c` is some collection (how ever it is implemented):

```
Iterator iter = c.iterator();
while (iter.hasNext())
{  Object obj = iter.next();
   // Do something to obj
}
```

  (Typically, you need to cast the result of `iter.next()`)

- Java 2's `Iterator` supercedes Java 1's `Enumeration`



### A program that reverses a list

This works but it's bad style:

```
public static ArrayList reverse(
                        ArrayList list)
{  ArrayList reverseList =
            new ArrayList(list.size());
   for (int i = 0; i < list.size(); i++)
   {  Object nextObject = list.get(i);
      reverseList.add(0, nextObject);
   }
   return reverseList;
}
```

Convince yourself that this version is not easily changed to use a different implementation



### Use interfaces and iterators!

Write a better version:

Convince yourself that this version is easily changed to use a different implementation