Collections

What Kinds of Collections are Available?

Java includes many collections with different semantics. Each kinds of collection has both an *Interface* and at least one concrete implementation class.

The collections we will study are:

Collection Type	Class	Description
List	`ArrayList` `LinkedList`	Ordered collection where elements can be inserted or removed at any location. Duplicates are allowed.
Set	`HashSet` `TreeSet`	Collection that determines its own ordering of elements. Programmer cannot add element at particular location. No duplicates allowed.

Other types of collections in the Java SE API are Stack, Queue, Deque (double-ended queue), and Tree.

Examples

1. List of Coins

As in the Coin Purse lab, use a List to contain Coins since we may have multiple Coins with same value

```
List<Coin> money = new ArrayList<Coin>( );
Coin five = new Coin(5);
Coin ten = new Coin(10);
money.add( five );
money.add( ten );
money.add( five ); // duplicate
// can add a new coin at any location
money.add( 0, new Coin(1) );
// Test if list has a "10" coins. Test uses Coin.equals()
Coin testcoin = new Coin(10);
if (money.contains(testcoin) ) money.remove( testcoin );
```

2. Set of Fruit

Create a set of the fruit we like.

```
Set<String> fruit = new HashSet<String>( );
fruit.add("Banana"); fruit.add("Apple"); fruit.add("Grape");
fruit.add("Orange");
// if you iterate ove the elements the order will not be same
for( String f: fruit ) System.out.println(f);
// can we add "Apple" again?
fruit.add("Apple"); // returns false -- already has Apple
fruit.contains("Durian"); // false -- I hate durian
fruit.remove("Grape");    // true
fruit.size();             // 3, only Apple, Banana, Orange
```

3. Map of key-value pairs

Map does not implement the *Collection* interface, but it is part of Java's collection framework.

A Map is a mapping (association) of keys to values. In Java, the keys and values can be object type.

Use a map to retrieve values by looking up their keys. For example, suppose we have different types of coupons. Red are worth 100Bt, Blue are 50Bt, Green 10Bt:

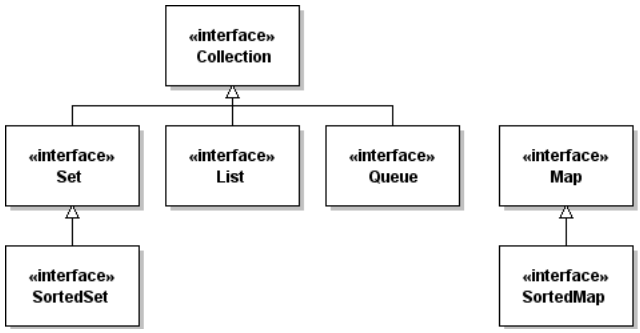
```
Map<String,Coupon> map = new HashMap<String,Coupon>( );
map.put("red", new Coupon(100) );
```

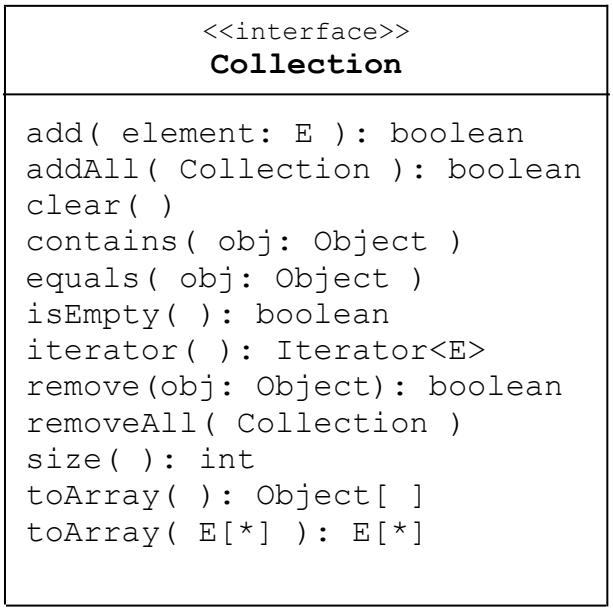
```
map.put("green", new Coupon(20) );
map.put("blue",  new Coupon(50) );
// now you can find coupons by color
if ( map.contains("blue") ) Coupon c = map.get("blue");
```

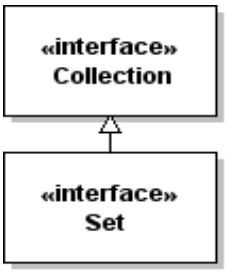
Core Collection Interfaces

The collections classes all implement a small number of interfaces. If you remember the interface methods, it will be much easier to use the collections.

The Collection interface is the base type for most collections. Collection defines most of the methods in Set, List, and Queue.







Collection has an iterator method, which means we can *iterate* over the elements in any collection using a *while* loop or *for-each* loop.

In the Set example above, the `for(String f: fruit)...` statement uses the Iterator to create a for-each loop. We can do the same thing directly using the Iterator:

```
Iterator<String> iterator = fruit.iterator();
while( iterator.hasNext() )
   System.out.println( iterator.next() ); // prints each fruit
```

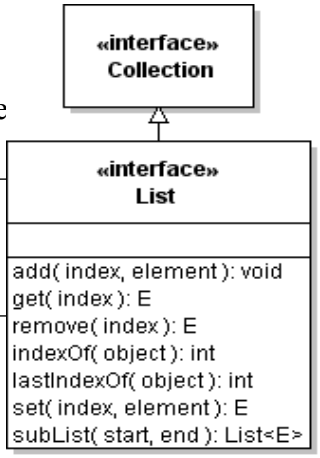
Set InterfaceThe *Set* interface does not add any new methods to *Collection*, but *Set* specifies a stronger *contract* for some of the methods. In particular:

add(element) succeeds only if element isn't in the set already.

equals(Object) is true if Object is a Set and both sets contain the same elements, even if the two Sets are instances of different Set classes.

List Interface

The *List* interface adds to *Collection* the ability to insert and remove elements at a spe (index) in a List. List also has methods to *find* the index of an element in the list.




```
public interface List<E> extends Collection<E> {
    // access by index
    E get(int index);
    // add at index (optional)
```

```
    void add(int index, E element);
    // remove by index (optional)
    E remove(int index);
    // search for element
    int indexOf(Object o);
    int lastIndexOf(Object o);
    // extra Iterator methods
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    // Range-view (not a copy) of this list
    List<E> subList(int from, int to);
}
```

Suppose we have a *List* of fruit names:

```
List<String> fruit = new ArrayList<String>( );
fruit.add("Apple"); fruit.add("Orange"); fruit.add("Grape");
fruit.add("Cake");
// Does fruit contain Cake?
int index = fruit.indexOf("Cake");
// Cake is not a fruit, so remove it
if (index >= 0) fruit.remove(index);
if (! fruit.contains("Cake")) System.out.println("no more Cake");
```

The subList method can be used to perform operations on a section of a List. For example, to remove elements 2 - 4 (assuming list has enough elements):

```
fruit.subList(2, 5).clear( ); // delete elements from fruit
```

Map Interface

Map provides an association between keys and values. Unlike other collections, Map has 2 type parameters: one for the key type and one for the value type.

For example, to mapping of Integer to String, you would use: Map<Integer,String>

The most common methods are shown here:

<<interface>> Map	
`clear( )`	
`containsKey( key ): boolean`	
`get(key: Object): Value`	get the value of a given *key*
`put(key: Key, value: Value)`	insert (*key, value*) into the map
`keySet( ): Set<Key>`	get all the keys in the map
`remove( key ): Value`	remove a (key, value) pair from map
`size( ): int`	number of keys in the map
`values( ): Collection<Value>`	get all the values in the map

Suppose we want to be able to convert *words* to *integers*, so if we see "eleven" in the input we know the value is 11. Use a map whose keys are the strings and whose values are integers:

```
Map<String,Integer> numbers = new HashMap<String,Integer>( );
numbers.put("one", 1);  // autoboxing: 1 --> new Integer(1)
numbers.put("two", 2);
numbers.put("three", 3);
...
numbers.put("twenty", 20);
```

When we process a word from the input we can look for it in the map and get the value:

```
// get a word from the input (scanner), test if it is a number
String word = scanner.next();
if ( numbers.contains(word) ) value = numbers.get(word);
else /* not a number */;
```

Collection Classes

Each Collection interface has at least one concrete class. There are also *abstract classes* which provide common code and simplify writing new collections.

ArrayList and LinkedList

ArrayList uses an array for storage and is usually faster for sorting and searching. LinkedList is usually faster if you frequently insert and remove element in the List.

If an ArrayList becomes full, it creates a new array and *copies* all the elements from old array to new array. If you know (approximately) how many elements will be in the ArrayList you can avoid this by reserving an initial capacity. For example, in the Coin Purse constructor, we can create an ArrayList with enough capacity so that the array is never copied:

```
   /** initialize a new Purse with given capacity */
   public Purse(int capacity) {
      money = new ArrayList<Valuable>( capacity );
      this.capacity = capacity;
```

HashSet and TreeSet

HashSet uses a hash table to store elements and is usually faster than TreeSet. It relies on the hash code of the elements to locate them. Therefore, the hashCode() method of objects in a HashSet should be something that doesn't normally change.

HashMap, Hashtable, and Properties

HashMap is the usual implementation of Map. Hashtable is an older implementation of a map that has extra methods, such as keys() and contains(key). Map has keySet and containsKey() for these.

A java.util.Properties object used to access properties of an application or system properties. Properties has methods to read key-values (as Strings) from a file and save them in a file. Java has a System Properties object that lets you find information about the operating environment.

Here are some examples:

```
Properties props = System.getProperties();
System.out.println("User is " + props.get("user.name") );
System.out.println("Your OS is " + props.get("os.name") );
```

To print all the System properties you *could* use the keySet method:

```
Properties props = System.getProperties();
Set keys = props.keySet( );
for(Object key: keys)
   System.out.printf("%s = %s\n", key, props.get(key) );
```

The Properties class has a *convenience method* list(OuputStream) that does the same thing:

```
   Properties props = System.getProperties( );
   props.list(System.out);
```

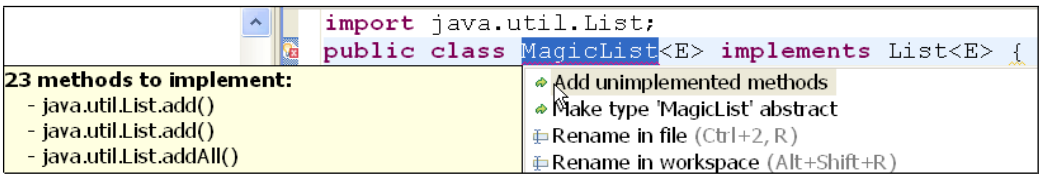
Applications use Properties to load application configuration data from a file.

Abstract Classes for Collections

Suppose you want to write your own List class. In an IDE you write:

```
public class MagicList<E> implements List<E> {
```

The IDE would then inform you that you must implement **23 methods**.



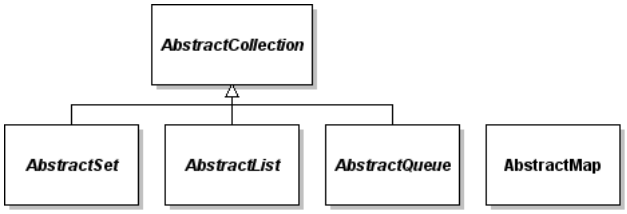
Instead, if you extend *AbstractList* (which implements *List*, of course):

```
public class MagicList<E> extends AbstractList<E> {
```

then you are only *required* to implement **2 method**s: get( ) and size().

To create a useful list, you'd also want to implement add(element), add(index,element), and a few other methods. Still, you can avoid a lot of coding by inheriting methods from AbstractList.

Java provides abstract collection classes that parallel the collection interfaces:



java.util.Collections is not a Collection

The java.util.Collections class (ends with "s") contains **static utility methods** for collections, such as sorting and searching, and creating an **immutable** view of a List, Set, or Map. The immutable views are *wrappers* not *copies*, and very useful if you want a method to return a collection without breaking encapsulation of the attribute that the collection represents.

Sorting and Sorted Collections

The only collections that the programmer can sort are Lists. To sort a List of objects that implement *Comparable* use: Collections.sort(*list*).

Some collections maintain sorted order automatically. A SortedSet always adds elements in sort order, either using the *natural order* (the *compareTo* method of *Comparable* objects) or using a *Comparator* object. Since String implements *Comparable*, we can maintain a sorted set of fruit names:

```
SortedSet<String> fruit = new TreeSet<String>( );
fruit.add("Orange");
fruit.add("Banana");
fruit.add("Apple");
for( String f: fruit ) System.out.print(f + " ");
// prints "Apple Banana Orange "
```