Collections and Generics

Collections

What is the Collections framework?

- Collections framework provides two things:
  - implementations of common high-level *data structures*: e.g. Maps, Sets, Lists, etc.
  - An organised class hierarchy with rules/formality for adding new implementations

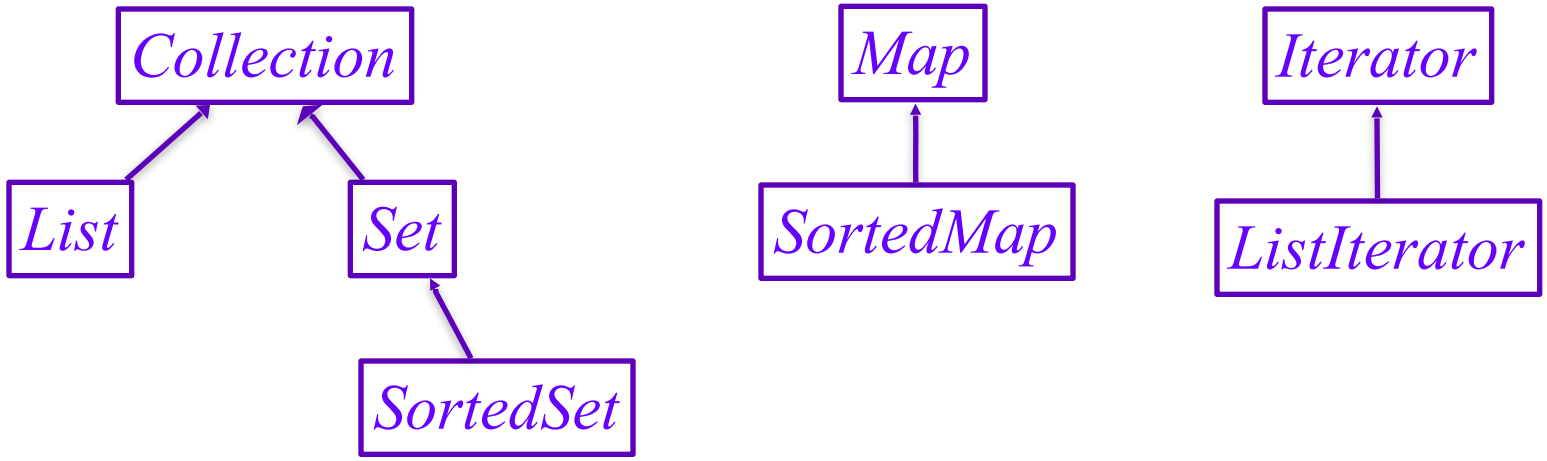
History

- Pre Java SDK1.2, Java provided a handful of data structures:
  - Hashtable
  - Vector
  - Bitset
- These were for the most part good and easy to use, but they were not organized into a more general framework.
- SDK1.2 added the larger skeleton which organizes a much more general set of data structures.
- Legacy datastructures retrofitted to new model.
- *Generic types*/autoboxing added in 1.5

General comments about data structures

- "Containers" for storing data.
- Different data structures provide different abstractions for getting/setting elements of data.
  - linked lists
  - hashtables
  - vectors
  - arrays
- Same data structures can even be implemented in different ways for performance/memory:
  - queue over linked list
  - queue over arrays

Collections-related Interface hierarchy



- The *Collection* inteface stores groups of Objects, with duplicates allowed
- The *Set* interface extends *Collection* but forbids duplicates
- The *List* interface extends *Collection*, allows duplicates, and introduces positional indexing.
- *Map* is a separate hierarchy

Collection implementations

- Note that Java does not provide any direct implementations of *Collection*.
- Rather, concrete implementations are based on other interfaces which extend Collection, such as Set, List, etc.
- Still, the most general code will be written using Collection to type variables.

A Peek at generics

Old way

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

New way with Generics …

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Another example of Generics

Here is a simple example taken from the existing Collections tutorial:

```
// Removes 4-letter words from c. Elements must be strings
static void expurgate(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext(); )
    if (((String) i.next()).length() == 4) i.remove();
}
```

Here is the same example modified to use generics:

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
  for (Iterator<String> i = c.iterator(); i.hasNext(); )
    if (i.next().length() == 4) i.remove();
}
```

Think "Collection of Strings"

Collection Interface

```
boolean add(Object o);
boolean addAll(Collection c);
void clear();
boolean contains(Object o);
boolean containsAll(Collection c);
boolean equals(Object o);
int hashCode();
boolean isEmpty();
Iterator iterator();
boolean remove(Object o);
boolean removeAll(Collection c);
boolean retainAll(Collection c);
int size();
Object[] toArray();
Object[] toArray(Object[] a);
```

Optional operation, throw UnsupportedOperationException

What does this mean in terms of what we've learned about Interfaces and OO architecture?

Comments on Collection methods

- Note the iterator() method, which returns an Object which implements the *Iterator* interface.
- *Iterator* objects are used to traverse elements of the collection in their natural order.
- Iterator has the following methods:
  - boolean hasNext(); // are there any more elements?
  - Object next(); // return the next element
  - void remove(); // remove the element returned after lest next()

Lets Try This

- Create a list of 100 sequential numbers.. now iterate through them and remove the numbers divisible by 5

List interface

- An interface that extends the Collections interface.
- An ordered collection (also known as a *sequence*).
  - The user of this interface has precise control over where in the list each element is inserted.
  - The user can access elements by their integer index (position in the list), and search for elements in the list.
- Unlike *Set*, allows duplicate elements.
- Provides a special *Iterator* called *ListIterator* for looping through elements of the List.

Additional methods in *List* Interface

- List extends Collection with additional methods for performing index-based operations:
  - void add(int index, Object element)
  - boolean addAll(int index, Collection collection)
  - Object get(int index)
  - int indexOf(Object element)
  - int lastIndexOf(Object element)
  - Object remove(int index)
  - Object set(int index, Object element)

List/ListIterator Interface

- The List interface also provides for working with a subset of the collection, as well as iterating through the entire list in a position friendly manner:
  - ListIterator listIterator()
  - ListIterator listIterator(int startIndex)
  - List subList(int fromIndex, int toIndex)
- *ListIterator* extends *Iterator* and adds methods for bi-directional traversal as well as adding/removing elements from the underlying collection.

Concrete List Implementations

- There are two concrete implementations of the *List* interface
  - LinkedList
  - ArrayList
- Which is best to use depends on specific needs.

LinkedList Class

- The LinkedList class offers a few additional methods for directly manipulating the ends of the list:
  - void addFirst(Object)
  - void addLast(Object);
  - Object getFirst();
  - Object getLast();
  - Object removeFirst();
  - Object removeLast();
- These methods make it natural to implement other simpler data structures, like Stacks and Queues.

Stack class

- **Stack**()
  Creates an empty Stack. **Method**
- boolean **empty**()
  Tests if this stack is empty.
- E **peek**()
  Looks at the object at the top of this stack without removing it from the stack.
- E **pop**()
  Removes the object at the top of this stack and returns that object as the value of this function.
- E **push**(E item)
  Pushes an item onto the top of this stack.
- int **search**(Object o)
  Returns the 1-based position where an object is on this stack.

Lets Try This

- Compare ArrayList and LinkedList performance for the following
  - Adding elements in the center, start, end
  - Accessing elements at start and end

Set Interface

- Set also extends *Collection*, but it prohibits duplicate items (this is what defines a Set).
- No new methods are introduced; specifically, none for index-based operations (elements of Sets are not ordered).
- Concrete Set implementations contain methods that forbid adding two equal Objects.
- More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element
- Java has two implementations: HashSet, TreeSet

Using Sets to find duplicate elements

```
import java.util.*;

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

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

HashSets and hash tables

- Lists allow for ordered elements, but searching them is very slow.
- Can speed up search tremendously if you don't care about ordering.
- Hash tables let you do this. Drawback is that you have no control over how elements are ordered.
- hashCode() computes integer (quickly) which corresponds to position in hash table.
- Independent of other objects in table.

Lets try this

- Build a collection of 10 unique car objects. Add an id to the car to establish uniqueness.

Tree Sets

- Another concrete set implementation in Java is TreeSet.
- Similar to HashSet, but one advantage:
  - While elements are added with no regard for order, they are returned (via iterator) in sorted order.
  - What is sorted order?
    - this is defined either by having class implement *Comparable* interface, or passing a *Comparator* object to the TreeSet Constructor.
    - Latter is more flexible: doesn't lock in specific sorting rule, for example. Collection could be sorted in one place by name, another by age, etc.

Comparable interface

- Many java classes already implement this. Try String, Character, Integer, etc.
- Your own classes will have to do this explicitly:
  - *Comparable* defines the method
    public int compareTo(Object other);
  - *Comparator* defines the method
    public int compare(Object a, Object b);
- As we discussed before, be aware of the general contracts of these interfaces.

Maps

- Maps are similar to collections but are actually represented by an entirely different class hierarchy.
- Maps store objects by key/value pairs:
  - map.add(“1234”, “Andrew”);
  - ie Object Andrew is stored by Object key 1234
- Keys may not be duplicated
- Each key may map to only one value

Map methods

- Here is a list of the Map methods:
  - void clear()
  - boolean containsKey(Object)
  - boolean containsValue(Object)
  - Set entrySet()
  - boolean get(Object)
  - boolean isEmpty()
  - Set keySet()
  - Object put(Object, Object)
  - void putall(Map)
  - Object remove(Object)
  - int size()
  - Collection values()

Map Implementations

- We won't go into too much detail on Maps.
- Java provides several common class implementations:
  - HashMap
    - a hashtable implementation of a map
    - good for quick searching where order doesn't matter
    - must override hashCode and equals
  - TreeMap
    - A tree implementation of a map
    - Good when natural ordering is required
    - Must be able to define ordering for added elements.

Generics

- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods
- Stronger type checks at compile time
  - Fixing compile-time errors is easier than fixing runtime errors

Elimination of Casts

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);

List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

Generic class

```
public class Box {
    private Object object;
    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

```
public class Box<T> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}

Box<Integer> integerBox = new Box<Integer>();
```

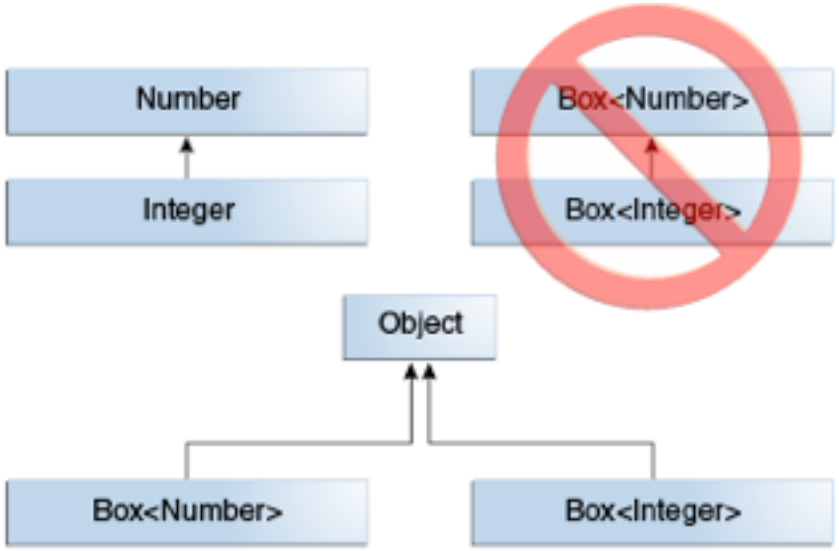
Generic Methods

```
public static <K, V> boolean join(K p1, V p2) {
    return p1.toString() +" "+ p2.toString();
}

Util.<Integer, String>join(p1, p2);
```

Generics & Inheritance

- Generics are implemented at the compiler level using erasure
  - Runtime system does not know about generics at all
  - Hence…
  - Instanceof wont work with generics



Effect of Erasure

```
List<Integer> mn = new ArrayList<>();
mn.add(5);
Collection n = mn;// A raw type - compiler throws
an unchecked warning
n.add("Hello");// Causes a ClassCastException to
be thrown.
Integer x = mn.get(1); //Problem!
```

Restrictions

- Cant use primitive type params
  - List<int> k;
- Cannot Create Instances of Type Parameters
  - public static <E> void append(List<E> list) {
  - E elem = new E(); // compile-time error
- Cannot Declare Static Fields Whose Types are Type Parameters
  - public class MobileDevice<T> {
  - private static T os;
- Cannot Use Casts or instanceof with Parameterized Types
  - if (list instanceof ArrayList<Integer>) { // compile-time error
- Cannot Create Arrays of Parameterized Types
  - List<Integer>[] arrayOfLists = new List<Integer>[2];
- Cannot Create, Catch, or Throw Objects of Parameterized Types
  - catch (T e) {
- Cannot overload
  - public void print(Set<String> strSet) { }
  - public void print(Set<Integer> intSet) { }