



Collections

Unit 09



Unit Objectives

- Understand the basic concepts of collections
- Explore the collection interfaces provided by Java
 - Interfaces
 - Abstract types
 - Concrete implementations
- Understand how the “legacy” classes and interfaces fit in with the more modern classes and interfaces

What Is a Collection?

- A collection is an object that groups multiple elements into a single unit
- Collections typically represent data items that form a natural group such as:
 - A poker hand
 - A collection of cards
 - A mail folder
 - A collection of letters
 - A telephone directory
 - A collection of name-to-phone-number mappings

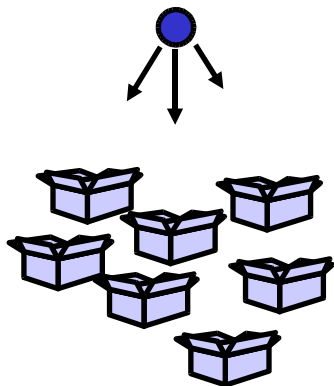


Collections Represent Data Structures

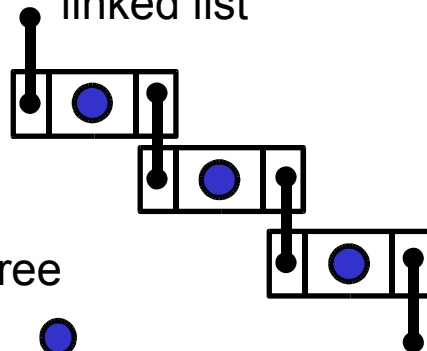
array - vector



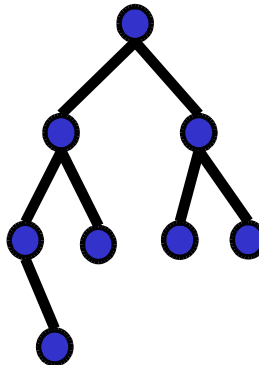
map - hash



linked list



tree



Specific Kinds of Collections

- Set

- Cannot contain duplicate elements

- e.g., employees, library books, processes running on a machine

- List

- Ordered collection, can contain duplicates

- e.g., webpage history, student roster

- Map

- Objects that maps keys to values, duplicate keys not allowed

- e.g., dictionary, property sheet

- Not a true collection interface

Arrays are also considered collections, though they are not part of the Collection Framework

The Java Collections Framework

- A Collections Framework is a unified architecture for representing and manipulating collections.
- It is comprised of three things:
 - Interfaces
 - Implementations
 - Algorithms

Interfaces and Implementations

- Interfaces – abstract data types representing collections
 - Allow collections to be manipulated independently of the details of their representations

 - Provide extension points
 - New collection types can be added which provide different implementations of the same method

- Implementations – concrete implementations of the collection interfaces
 - Reusable data structures

Algorithms

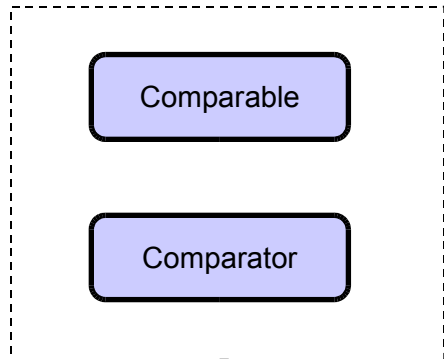
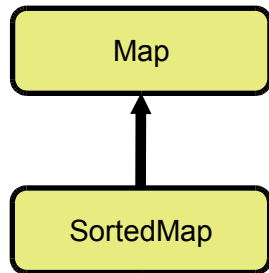
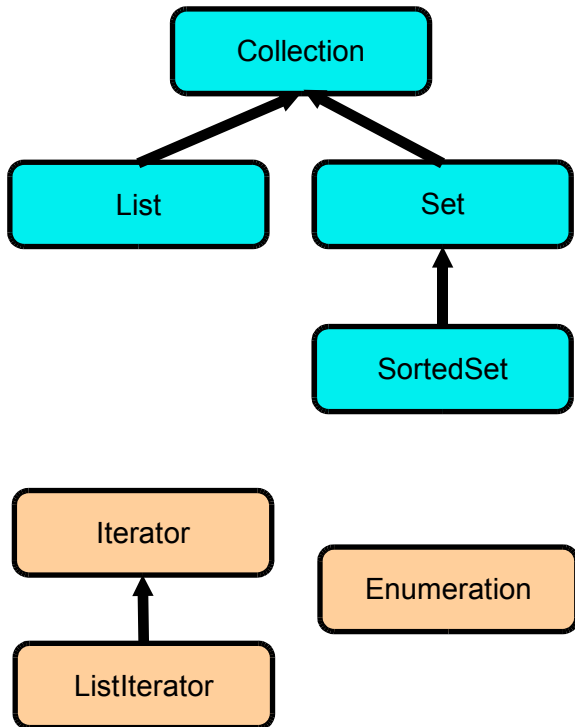
- Algorithms – methods that perform useful computations on objects that implement collection interfaces
 - e.g., searching and sorting
 - Reusable functionality via polymorphism
 - Same method can be used on many different implementations of the appropriate collections interface

Benefits of a Collections Framework

- Reduces
 - Programming effort
 - Effort to learn and use new APIs
 - Effort to design new APIs
- Increases program speed and quality
- Allows interoperability among unrelated APIs
- Encourages software reuse



Interfaces in the Framework



The Collection Interface

- Used to change a collection and pass them from one method to another

For example:

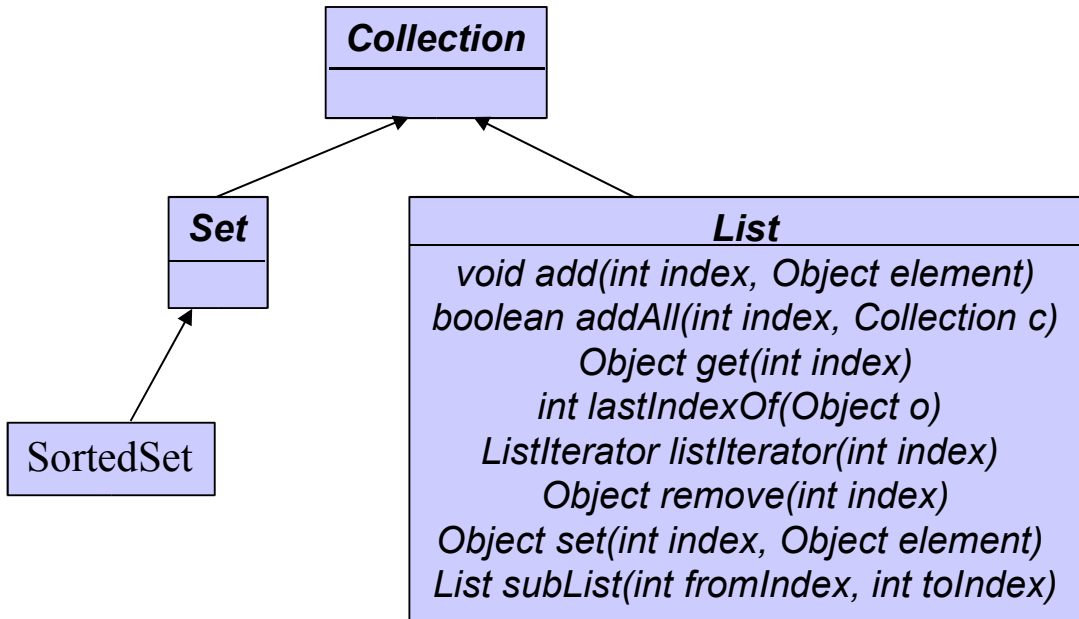
- Add to or remove from a collection
- Test membership

- Defines methods to
 - Facilitate looping through collections
 - Convert collections to arrays

Collection

```
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)
```

Collections, Sets and Lists



- The collection interfaces form a hierarchy

The Map Interface

- A Map maps keys to values
 - Cannot contain duplicate keys
- Defines the interface needed to manipulate such a collection:
 - Add/remove a key-value pair
 - Given a key, get the value
 - Test membership
- A map's contents can be viewed in one of three **collection views**,
 - A set of keys
 - A collection of values
 - A set of key-value mappings

Map

void clear()

boolean containsKey(Object key)

boolean containsValue(Object value)

Set entrySet()

boolean equals(Object o)

Object get(Object key)

int hashCode()

boolean isEmpty()

Set keySet()

Object put(Object key)

void putAll(Map t)

Object remove(Object key)

int size()

Collection values()

Comparing Objects

- To sort items in a collection, there must be a way to impose a **total ordering** on the items
 - For any two items in the collection, it must be possible to compare the objects and unambiguously determine whether:
 - Object A comes before object B
 - Object B comes before object A
 - Object A and object B are equal
 - There are two ways to order objects
 - The Comparable interface
 - The Comparator interface

More on Comparing Objects

- The **Comparable** Interface
 - Implements a class whose objects are capable of comparing other objects to themselves
 - Such classes are said to have a *natural ordering*
- The **Comparator** interface
 - Implements a class whose purpose is to compare other objects to each other
 - The same two objects may compare differently using different comparators

Comparable

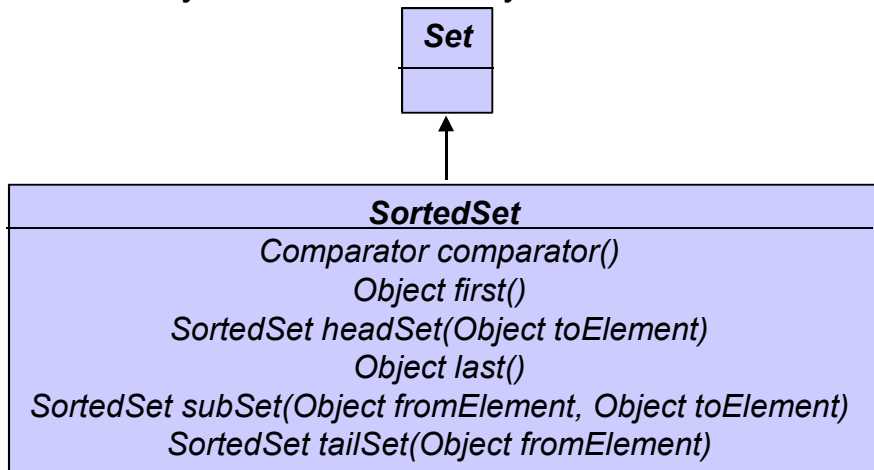
```
int compareTo(Object o)
```

Comparator

```
int compare(Object o1, Object o2)
```

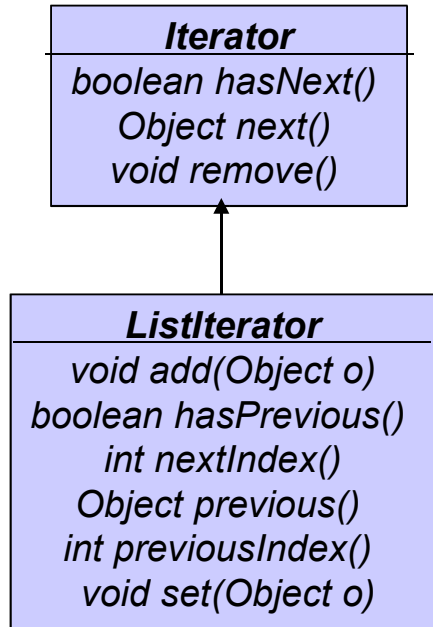
Sorted Collections

- **SortedSet** is a **Set** with an intrinsic (and automatically maintained) order
 - Additional methods expose this order
- **SortedMap** is a **Map** with similar properties, based on key order
- In either case, order may be determined by a natural order or a comparator



Iterators

- Iterators provide a convenient way to loop over the entire contents of a collection one at a time
- **ListIterator** adds methods that expose the sequential nature of the underlying list
- `add` and `remove` operations “pass through” to the underlying collection
- Iterators of sorted collections iterate through the collection according to its underlying order



The Iterator Code Pattern

```
Collection c;  
  
Iterator i = c.iterator();  
while (i.hasNext()) {  
    Object o = i.next();  
    // process this object  
}
```

Interfaces and Implementations

		IMPLEMENTATIONS				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Legacy
I N T E R F A C E S	Set	HashSet		TreeSet		
	List		ArrayList		LinkedList	Vector, Stack
	Map	HashMap		TreeMap		HashTable, Properties

Best practice: code to the interface,
not the implementation

Example

```
public class MapExample {
    public static void main(String args[]) {
        Map map = new HashMap();
        Integer ONE = new Integer(1);
        for (int i=0, n=args.length; i<n; i++) {
            String key = args[i];
            Integer frequency =(Integer)map.get(key);
            if (frequency == null) {
                frequency = ONE;
            } else {
                int value = frequency.intValue();
                frequency = new Integer(value + 1);
            }
            map.put(key, frequency);
        }
        System.out.println(map);
        Map sortedMap = new TreeMap(map);
        System.out.println(sortedMap);
    }
}
```

Implementation Choices

•Set / Map

–HashSet / HashMap

- Very fast, no ordering
- Choice of *initial capacity* and *load factor* important for performance

–TreeSet / TreeMap

- Maintains balanced tree, good for sorted iterations
- No tuning parameters

–HashTable

- Synchronized
- Be sure to use **Map** interface

•List

–ArrayList

- Very fast
- Can use native method **System.arraycopy**

–LinkedList

- Good for volatile collection, or adding to front (e.g., queues)

–Vector

- Synchronized
- Be sure to use **List** interface

Legacy Collections

- The legacy collection classes are still available, but their implementations have changed.



–java.util.Vector

- Extendable, shrinkable, indexed list



–java.util.Stack

- Extends Vector to allow push and pop on a LIFO



–java.util.BitSet

- Expandable set of True/False flags



–java.util.Dictionary

- Abstract class now obsolete and replaced by java.util.Map



–java.util.Hashtable

- Efficient storage of objects with no natural organization



–java.util.Properties

- Stores key-value pairs. The key is the name of a property.

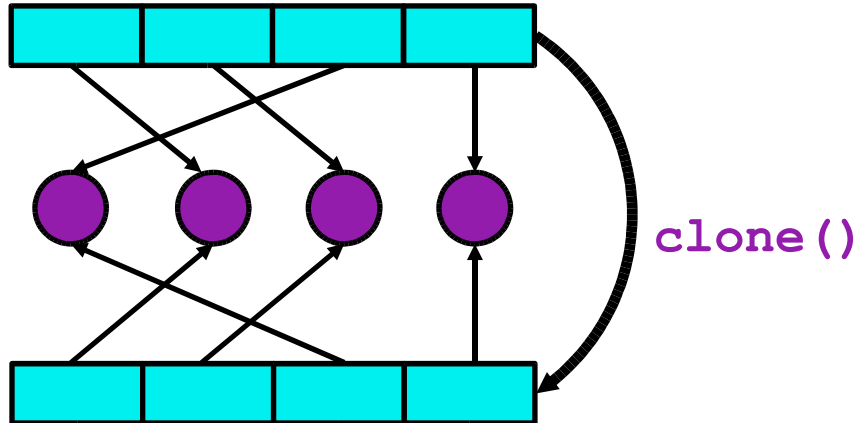
Cloning Collections

- You can make a copy of most collections with the `clone()` method
 - This creates a new collection but does not clone the objects stored in the collection (called a shallow copy)

Collection1

Stored
Objects

Collection2



The Collections Class

- `java.util.Collections` consists exclusively of static methods that operate on or return collections. It contains:
 - Polymorphic algorithms that operate on collections, e.g.,
 - `binarySearch`
 - `copy`
 - `min` and `max`
 - `replace`
 - `reverse`
 - `rotate`
 - `shuffle`
 - `sort`
 - `swap`
 - “Wrappers” – returns a new collection backed by a specified collection
 - Synchronized collections
 - Unmodifiable collections

What You Have Learned

- What collections are
- What the Java Collections Framework is
- Which interfaces support Collections
- Which concrete classes implement Collections
- How to use the **Collections** class