What is Java collections

• A collection is an object that represents a group of objects

• A number of pre-packaged implementations of common 'container' classes

• Collections API offers a structured paradigm for manipulating groups of objects in an ordered fashion, and employs design patterns (e.g., the iterator pattern) that make common programming tasks much simpler.
Why use Collections

- Reduces programming effort
- Increases performance
- Provides interoperability between unrelated APIs
- Reduces the effort required to learn APIs
- Reduces the effort required to design and implement APIs
- Fosters software reuse

Java Collections

- Several implementations associated with each of the basic interfaces
- Each has its own advantages/disadvantages
  - Lists
    - ArrayList, LinkedList
  - Sets
    - HashSet, SortedSet
  - Maps
    - HashMap, SortedMap

Set and List

- Set: No duplicate elements permitted. May or may not be ordered.
- List: A sequence. Duplicates are generally permitted.

Map

- Map: A mapping from keys to objects. Each key can also be an object
  - No duplicate keys permitted
  - The group of keys is a set

Collections -- classes

- Concrete classes that implement the interfaces
  - HashSet, SortedSet
  - ArrayList, LinkedList
  - HashMap, SortedMap
- Create a collection object: Generally hold references to the interface and not the specific class

List myList = new ArrayList();
List otherList = new ArrayList(5);
Map database = new HashMap();
Set things = new HashSet();

Java Collections -- Adding Items

- For Sets and Lists, use **add**
  
  ```java
  List myList = new ArrayList();
  myList.add("A String");
  myList.add("Other String");
  ```

- For Maps, use **put**
  
  ```java
  Map myMap = new HashMap();
  myMap.put("google", "http://www.google.com");
  myMap.put("yahoo", "http://www.yahoo.com");
  ```
Java Collections – Copying

- Very easy, just use addAll()

```java
List myList = new ArrayList(); // assume we add items to the list
List otherList = new ArrayList();
otherList.addAll(myList);
```

Collections – Getting Individual Items

- Use `get()`
- Note that we have to `cast` the object to its original type.

```java
String s = (String)myList.get(0); // get first element
String s2 = (String)myList.get(10); // get tenth element
```

- Sets and Lists...

```java
String s = (String)myList.get("google");
String s2 = (String)myMap.get("yahoo");
```

Collections – Getting all items

- For Lists, we could use a `for` loop, and loop through the list to `get()` each item.
- But this doesn’t work for Maps.
- To allow generic handling of collections, Java defines an object called an `Iterator`
  - An object whose function is to walk through a Collection of objects and provide access to each object in sequence.

```java
Set a_set = new HashSet();
Iterator itr = a_set.iterator();
while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
```

Collections – Getting all items

- Get an iterator using the `iterator()` method
- Iterator objects have three methods:
  - `next()` — gets the next item in the collection
  - `hasNext()` — tests whether it has reached the end
  - `remove()` — removes the item just returned
- Basic iterators only go forwards
  - Lists objects have a `ListIterator` that can go forward and backward

```java
List a_list = new ArrayList();
for(Iterator itr = a_list.iterator(); itr.hasNext(); ) {
    System.out.println(itr.next());
}
```

Iterator examples

```java
Set a_set = new HashSet();
Iterator itr = a_set.iterator();
while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
```

```
List a_list = new ArrayList();
for(Iterator itr = a_list.iterator(); itr.hasNext(); ) {
    System.out.println(itr.next());
}
```
Collections – Other Functions

• The java.util.Collections class has many useful methods for working with collections
  – min, max, sort, reverse, search, shuffle
• Virtually all require your objects to implement an extra interface, called Comparable

Collections – Comparable

• The Comparable interface labels objects that can be compared to one another
  – Allows sorting algorithms to be written to work on any kind of object
  – so long as they support this interface
• Single method to implement
  public int compareTo(Object o);
• Returns
  – A negative number if parameter is less than the object
  – Zero if they’re equal
  – A positive number if the parameter is greater than the object

Collections – Comparator

• Like Comparable, but is a stand-alone object used for comparing other objects
  – Useful when you want to use your criteria, not that of the implementor of the object.
  – Or altering the behaviour of a system
• Again has single method:
  public int compare(Object obj1, Object obj2)

Comparator Example

• In this example, String comparison method compareTo() is adopted to override compare()

public class AlphaComparison implements Comparator{
  public int compare(Object obj1, Object obj2){
    String s1 = ((String)obj1).toLowerCase();
    String s2 = ((String)obj2).toLowerCase();
    return s1.compareTo(s2);
  }
}

The Set Interface

• The Set interface is used to represent an unordered collection of objects. The two concrete classes in this category are HashSet and TreeSet.
• A set is in some ways a stripped-down version of a list. Both structures allow you to add and remove elements, but the set form does not offer any notion of index positions. All you can know is whether an object is present or absent from a set.
• The Collections Framework provides two general-purpose implementations of the Set interface: HashSet and TreeSet. The HashSet class is built on the idea of hashing; the TreeSet class is based on a structure called a binary tree.

HashSet and TreeSet

• More often than not, you will use a HashSet for storing your duplicate free collection.
• HashSet allows at most one null element.
• HashSet is faster than other implementations of Set (TreeSet and LinkedHashSet).
• The add method of Set returns false if you try to add a duplicate element.
• The TreeSet implementation is useful when you need to extract elements from a collection in a sorted manner.
• In order to work properly, elements added to a TreeSet must be sortable.
HashSet & TreeSet example

```java
import java.util.*;
public class SetExample {
  public static void main(String[] args) {
    Set set = new HashSet();
    set.add("Bernadine");
    set.add("Elizabeth");
    set.add("Gene");
    set.add("Elizabeth");
    set.add("Clara");
    System.out.println(set);
    Set sortedSet = new TreeSet(set);
    System.out.println(sortedSet);
  }
}
```

Output:
```
[Gene, Clara, Bernadine, Elizabeth]
[Bernadine, Clara, Elizabeth, Gene]
```

Set Operations

```java
public interface Set<E> extends Collection<E> {
  // Basic operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);
  boolean remove(Object element);
  Iterator<E> iterator();
  // Bulk operations
  boolean containsAll(Collection<? super E> c);
  boolean addAll(Collection<? extends E> c);
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
  void clear();
  // Array Operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

List interface

- A list is an ordered collection.
- Lists may contain duplicate elements.
- List interface includes operations for:
  - Positional access: manipulates elements based on their numerical position in the list
  - Search: searches for a specified object in the list and returns its numerical position
  - Iteration: extends Iterator semantics to take advantage of the list’s sequential nature
  - Range-view: performs arbitrary range operations on the list.

The List interface

List Operations

```java
public interface List<E> extends Collection<E> {
  // Positional access
  E get(int index);
  E set(int index, E element);
  boolean add(E element);
  void add(int index, E element);
  E remove(int index);
  boolean addAll(int index, Collection<? extends E> c);
  // Search
  int indexOf(Object o);
  int lastIndexOf(Object o);
  // Iteration
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);
  // Range-view
  List<E> subList(int from, int to);
}
```

List Iterators

- List’s iterator returns the elements of the list in proper sequence
- ListIterator allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.

```
List aList = new ArrayList();
aList.add("*");
aList.add("*");
aList.add("*");
aList.add("*");
ListIterator itr = aList.listIterator();
System.out.println(itr.previousIndex());
System.out.println(itr.nextIndex());
System.out.println(itr.previousIndex());
System.out.println(itr.nextIndex());
```

List Iterator example

```
List aList = new ArrayList();
aList.add("*");
aList.add("*");
aList.add("*");
aList.add("*");
ListIterator itr = aList.listIterator();
System.out.println(itr.previousIndex());
System.out.println(itr.nextIndex());
System.out.println(itr.previousIndex());
System.out.println(itr.nextIndex());
```

Output:
```
-1
0
0
1
```
**Map interface**

- A **Map** is an object that maps keys to values.
- A map cannot contain duplicate keys: Each key can map to at most one value.
- It models the mathematical *function* abstraction.

**Map Operations**

- The alteration operations allow you to add and remove key-value pairs from the map. Both the key and value can be null. However, you should not add a Map to itself as a key or value.
  - Object put(Object key, Object value)
  - Object remove(Object key)
  - void putAll(Map mapping)
  - void clear()

**Map Operations**

- The query operations allow you to check on the contents of the map:
  - Object get(Object key)
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
  - int size()
  - boolean isEmpty()

**Map Operations**

- The last set of methods allow you to work with the group of keys or values as a collection.
  - public Set keySet()
  - public Collection values()
  - public Set entrySet()

- Because the collection of keys in a map must be unique, you get a Set back. Because the collection of values in a map may not be unique, you get a Collection back.

**Map Iteration**

- No direct iteration over Maps
  - Maps do not provide an iterator() method as do Lists and Sets
- Three indirect ways: iteration over Sets, Collections, or key-value pairs
  - Get a Set of keys by keySet();
  - Get a Collection of values by values();
  - Get a Set of key-value pairs entrySet();
Map Iterator example

```java
import java.util.*;
public class IterateValuesOfHashMapExample {
    public static void main(String[] args) {
        HashMap hMap = new HashMap();
        hMap.put("1","One");
        hMap.put("2","Two");
        hMap.put("3","Three");
        Collection c = hMap.values();
        Iterator itr = c.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
    }
}
```

Output:
```
Three
Two
One
```

Map Iterator example (II)

```java
import java.util.*;
public class IterateValuesOfHashMapExample {
    public static void main(String[] args) {
        Map data = new HashMap();
        data.put(OK, "HTTP_OK");
        data.put(FORBIDDEN, "HTTP_FORBIDDEN");
        data.put(NOT_FOUND, "HTTP_NOT_FOUND");
        Set entries = data.entrySet();
        for(Map.Entry entry : entries) {
            Object key = entry.getKey();
            Object value = entry.getValue();
            System.out.println(key + " = " + value);
        }
    }
}
```

Output:
```
OK = HTTP_OK
FORBIDDEN = HTTP_FORBIDDEN
NOT_FOUND = HTTP_NOT_FOUND
```

HashMap & TreeMap

- The HashMap and TreeMap classes are two concrete implementations of the Map interface.
- The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.
- The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.
- Depending upon the size of your collection, it may be faster to add elements to a HashMap, then convert the map to a TreeMap for sorted key traversal.
- With the TreeMap implementation, elements added to the map must be sortable.

Example: HashMap & TreeMap

```java
import java.util.*;
public class TestMap {
    public static void main(String[] args) {
        // Create a HashMap
        Map hashMap = new HashMap();
        hashMap.put("Smith", 30);
        hashMap.put("Anderson", 31);
        hashMap.put("Lewis", 29);
        hashMap.put("Cook", 29);
        System.out.println("Display entries in HashMap");
        System.out.println(hashMap);
        // Create a TreeMap from the previous HashMap
        Map treeMap = new TreeMap(hashMap);
        System.out.println("Display entries in ascending order of key");
        System.out.println(treeMap);
    }
}
```

Output:
```
Display entries in HashMap
{Smith=30, Lewis=29, Anderson=31, Cook=29}
Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}
```

Exercise

```java
import java.util.*;
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap hm = new HashMap();
        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Todd Hall", new Double(99.22));
        TreeMap tm = new TreeMap(hm);
        System.out.println(tm);
        Set set = tm.entrySet();
        Iterator i = set.iterator();
        while(i.hasNext()) {
            System.out.print(i.next() + ":");
            System.out.println(tm.get(i.next()));
        }
        collection col = tm.values();
        i = col.iterator();
        while(i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```
Output

{Jane Baker=1378.00, John Doe=3434.34, Todd Hall=99.22, Tom Smith=123.22}
Jane Baker: 3434.34
Todd Hall: 123.22
1378.00
3434.34
99.22
123.22