COLLECTION FRAMEWORK

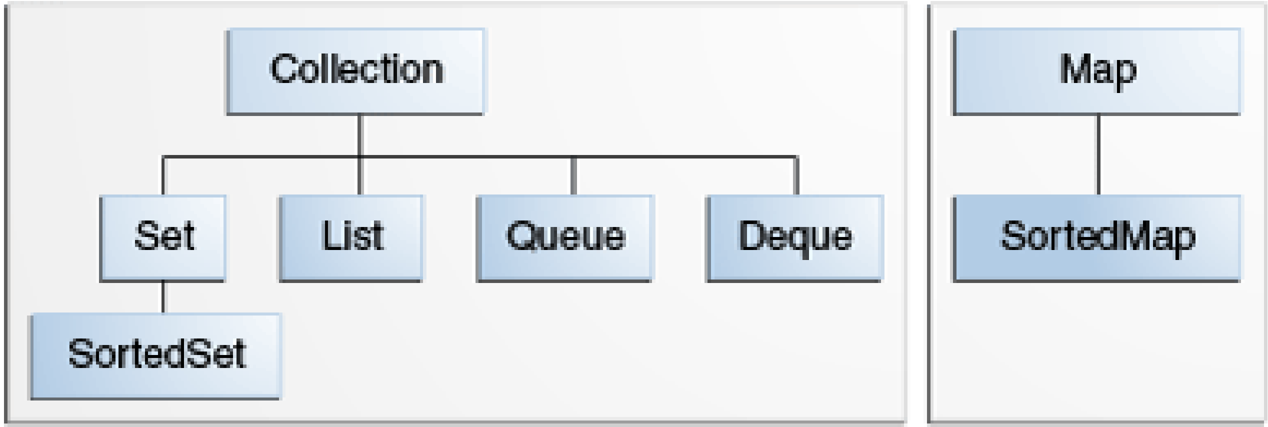
Introduction to Collections

- A **collection** - sometimes called a container - is simply an object that groups multiple elements into a single unit.
  - poker hand (a collection of cards), a mail folder (a collection of letters), a telephone directory (a mapping of names to phone numbers)

- A **collections framework** is a unified architecture for representing and manipulating collections.
  - **Interfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.
  - **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
  - **Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Core Collection Interfaces

- Encapsulating different types of collections
- The foundation of the Java Collections Framework (import java.util.*)
- All Generic interfaces



- ***Collection*** — the root of the collection hierarchy. A collection represents a group of objects known as its elements
- ***Set*** — a collection that cannot contain duplicate elements.
- ***List*** — an ordered collection (sometimes called a sequence)
- ***Queue*** — a collection used to hold multiple elements prior to processing
- ***Map*** — an object that maps keys to values. A Map cannot contain duplicate keys

http://docs.oracle.com/javase/tutorial/collections/index.html
http://www.tutorialspoint.com/java/java_collections.htm

Collection Interfaces

- A Collection represents a group of objects known as its elements
  - Conversion Constructor

```
Collection<String> c;
List<String> list = new ArrayList<String>(c);
```

  - Base Operations
    - size(), isEmpty(), contains(), add(), remove(), iterator()
  - Traversing Collections

```
//aggregate operators

myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e));
```

```
//enhanced for

for (Object o : collection)
    System.out.println(o);
```

```
//Iterators
while ( it.hasNext() ) {
    if (!condition(it.next()))
        it.remove();
}
```

  - Collection Interface Bulk Operations
    - containAll(), allAll(), removeAll(), clear()
  - Collection Interface Array Operations
    - String[] a = c.toArray(new String[0]);

Modifier and Type	Method and Description
boolean	add(E e)Adds the specified element to this set if it is not already present (optional operation).
boolean	addAll(Collection<? extends E> c)Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	clear()Removes all of the elements from this set (optional operation).
boolean	contains(Object o)Returns true if this set contains the specified element.
boolean	containsAll(Collection<?> c)Returns true if this set contains all of the elements of the specified collection.
boolean	equals(Object o)Compares the specified object with this set for equality.
int	hashCode()Returns the hash code value for this set.
boolean	isEmpty()Returns true if this set contains no elements.
Iterator<E>	iterator()Returns an iterator over the elements in this set.
boolean	remove(Object o)Removes the specified element from this set if it is present (optional operation).
boolean	removeAll(Collection<?> c)Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	retainAll(Collection<?> c)Retains only the elements in this set that are contained in the specified collection (optional operation).
int	size()Returns the number of elements in this set (its cardinality).
Object[]	toArray()Returns an array containing all of the elements in this set.
<T> T[]	toArray(T[] a)Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Collection Implementations

- Classes that implement the collection interfaces typically have names in the form of <Implementation-style><Interface>
- The general-purpose implementations support all of the optional operations in the collection interfaces and have no restrictions on the elements they may contain.

Interfaces	**Implementations**				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	**HashSet**		TreeSet		LinkedHashSet
List		**ArrayList**		LinkedList	
Deque		ArrayDeque		**LinkedList**	
Map	**HashMap**		TreeMap		LinkedHashMap

Collection : Set

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<>(); // or TreeSet<String>()
        for ( final String a : args )
            if ( !s.add(a) ) System.out.println("Duplicate detected: " + a);

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

Now run the program.

```
    % java FindDups i came i saw i left
```

The following output is produced.

```
    Duplicate detected: I
    Duplicate detected: I
    4 distinct words: [i, left, saw, came]
```

Collection : List

```
import java.util.*;
public class ListExample {
  public static void main(String[] args) {
    List<String> names = new ArrayList<>() ; // or LinkedList<>()

    // add, allAll
    names.add("Park") ;
    names.add("Kim") ;

    // toString
    System.out.println(names.toString()) ; // [Park, Kim]

    // add
    names.add(1, "Lee") ;

    // size, get
    for ( int i = 0 ; i < names.size() ; i ++ ) System.out.println(names.get(i)) ;
    // Park
    // Lee
    // Kim
```

```
        // remove
        names.remove("Kim") ; // remove(int index), removeAll(Collection<?> c)
        // indexOf
        int foundIndex = names.indexOf("Kim") ; // lastIndexOf also supported
        if (  foundIndex == -1 ) // ! names.contains(“Kim”), containsAll()
            System.out.println("Kim not Found") ; // Kim not Found
        else {
            System.out.println("Kim Found") ;
            names.remove(foundIndex)
        }

        // subList, clear
        names.subList(0, 1).clear(); // Remove Park

        // Iterator
        Iterator<String> it = names.iterator() ;
        while ( it.hasNext() ) System.out.println(it.next()) ;
        // Lee

        // clear, isEmpty
        names.clear();
        assert ( names.isEmpty() == true );
    }
}
```

Collection : Queue

```
import java.util.*;

public class Countdown {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();  //FIFO

        for (int i = time; i >= 0; i--) queue.add(i);     // offer(i)
        for (int i = time; i >= 0; i--) System.out.println(queue.element());

        while (!queue.isEmpty()) {
            System.out.println(queue.remove());     //poll()
            Thread.sleep(1000);
        }
    }
}
```

Collection : Map

```
import java.util.*;
public class MapExample {
  public static void main(String[] args) {
    Map<String, Integer> cityPopulation = new HashMap<>() ;

    cityPopulation.put("Busan", 350) ; // putAll
    cityPopulation.put("Seoul", 1000) ;
    cityPopulation.put("Daejon", 150) ;

    System.out.println(cityPopulation) ; // {Busan=350, Seoul=1000, Daejon=150}

    if ( cityPopulation.containsKey("Daejon") )
       System.out.println(cityPopulation.get("Daejon")) ; // 150

    cityPopulation.remove("Daejon") ;

    Set<String> cities = cityPopulation.keySet() ;
    System.out.println(cities) ; // [Busan, Seoul]

    Collection<Integer> population = cityPopulation.values() ;
    System.out.println(population) ; // [350, 1000]
```

```
        cityPopulation.replace("Busan", 300);
        for ( final String key : cityPopulation.keySet() ) {
            System.out.println( String.format("키 : %s, 값 : %s", key, cityPopulation.get(key)) );
        }

        Iterator<String> keys = cityPopulation.keySet().iterator();
        while ( keys.hasNext() ) {
            String key = keys.next();
            System.out.println( String.format("키 : %s, 값 : %s", key, cityPopulation.get(key)) );
        }

        for ( final Map.Entry<String, Integer> elem : cityPopulation.entrySet() ) {
            System.out.println( String.format("키 : %s, 값 : %s", elem.getKey(), elem.getValue()) );
        }

    }
}
```

```
키 : Busan, 값 : 300
키 : Seoul, 값 : 1000
키 : Busan, 값 : 300
키 : Seoul, 값 : 1000
키 : Busan, 값 : 300
키 : Seoul, 값 : 1000
```

Benefits

- Reduces programming effort
  - By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program.
- Increases program speed and quality
  - This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
- Allows interoperability among unrelated APIs
  - e.g., Network API or GUI toolkit uses a collection of node or column names.
- Reduces effort to learn and to use new APIs
  - With the advent of standard collection interfaces, many problems went away.
- Reduces effort to design new APIs
  - Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections.
- Fosters software reuse
  - New data structures that conform to the standard collection interfaces are by nature reusable.

Q&A