**Java Collections**

**A Collection:**

A ***collection*** is a collection of objects that share some common relationship or purpose. Different types of collections have specific names and expected behaviors. These include arrays, lists, vectors, sets, queues, tables, dictionaries, and maps (Data structure that maps keys to values).

These collections are differentiated by how they store objects in memory, how objects are retrieved, and ordered, and whether they allow null values or duplicate entries.

**A collection Framework:**

Oracle’s Java documentation describes its collection framework as:

“A unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details”

A collection framework refer to the entire set of interfaces while a collections class is a java.util class with in the framework

***Collection framework is based on Interfaces***

**What’s in the framework, what’s not?**

Arrays and the Array utilities in the java.util.Arrays class are not considered part of this framework. A collection objects implements the Collection interface, with the exception of maps. The Collection interface is the foundation of the collection hierarchy in Java.

**The Collection Interface**

This interface is often used when you want to **pass collections** around and manipulate then with **maximum flexibility and generality**.

***collection interface doesn’t have sort()***

***List interface extends collection. Collection is a base interface***

**What’s a polymorphic algorithm**

Oracles’s documentation describes a polymorphic algorithm as a piece of reusable functionality.

**The Collection Class:**

Its important to understand that this Collections class is not the Collections Framework.

The framework contains many interfaces and implemented classes, as well as helper classes, for which this Collections class is just one example.

At one time, Java had interfaces, but no support for static or default methods on the, so useful methods were packaged in these helper classes.

Some of these methods have since been implemented on the interfaces themselves, but there’s still some functionality on the Collections class we might find useful.

**addAll()**

Collections.*addAll*(cards, cardArray);

*1st argument is the list to add the elements to  
2nd argument for the elements to be added, but unlike list's addAll method, this is a variable argument of elements to be added*

**Copy()**

**This methods copies elements from one list to another**

Collections.*copy*(cards, kingsOfClubs);

*1st argument is the destination of the copied elements.  
2nd argument is the elements to be copied  
you cannot use this method if the number of elements in the current list (kingsOfClubs) is less than the number of the source list (cards)*

If you want the full list copy, use List.copyOf(); but this will be unmodifiable class.

**Collections.shuffle(), Collections.reverse()**

Collections.*shuffle*(deck1);  
Card.*printDeck*(deck1, **"Shuffled Deck"**, 4);  
  
Collections.*reverse*(deck1);  
Card.*printDeck*(deck1, **"Reversed Deck"**, 4); //reverse the shuffled deck

**.sort()**

**var** sortingAlgorithm = Comparator.*comparing*(Card::rank)  
 .thenComparing(Card::suit);  
Collections.*sort*(deck, sortingAlgorithm);  
Card.*printDeck*(deck, **"Standard Deck sorted by rank, suit"**, 13);  
Collections.*reverse*(deck);  
Card.*printDeck*(deck, **"Sorted by rank, suit reversed"**, 13);

**.sublist, indexOfSublist(), disjoint()**

*//methods that compare full list to sublist  
//carving out a couple of smaller lists from the standard deck and passing them to a new ArrayList constructor*List<Card> kings = **new** ArrayList<>(deck.subList(4, 8));  
Card.*printDeck*(kings, **"Kings in deck"**, 1);  
  
List<Card> tens = **new** ArrayList<>(deck.subList(16, 20));  
Card.*printDeck*(tens, **"tens in deck"**, 1);  
  
*//returns an int if sublist is present in the list or -1 if it isnt***int** subListIndex = Collections.*indexOfSubList*(deck, tens);  
System.***out***.println(**"sublist index for tens = "** + subListIndex);

System.***out***.println(**"Contains = "** + deck.containsAll(tens));// returns true

*//disjoint method return true if the two collections dont share elements, or false if they do***boolean** disjoint = Collections.*disjoint*(deck, tens);  
System.***out***.println(**"disjoint = "** + disjoint);*// return false***boolean** disjoint2 = Collections.*disjoint*(kings, tens);  
System.***out***.println(**"disjoint = "** + disjoint2);*// returns true*

**Binary Search on List**

*//binary search on Lists  
//list should be sorted*deck.sort(sortingAlgorithm);  
Card tensOfHearts = Card.*getNumericCard*(Card.Suit.***HEART***, 10);  
**int** foundIndex = Collections.*binarySearch*(deck, tensOfHearts, sortingAlgorithm);  
System.***out***.println(**"foundIndex = "** + foundIndex);//foundIndex = 34  
System.***out***.println(deck.get(foundIndex));// 10♥(8)

*//we can acquire the same result evewn without sorting our list by using list's indexOf()*System.***out***.println(**"foundIndex = "** + deck.indexOf(tensOfHearts));//foundIndex = 34

**replaceAll()**

*// collections replaceAll()  
// it requires to replace 1 or more elements in the list  
// this method returns a boolean value, true if one or more elements was replaced or false if not*Card tensOfClubs = Card.*getNumericCard*(Card.Suit.***CLUB***, 10);  
Collections.*replaceAll*(deck, tensOfClubs, tensOfHearts);  
Card.*printDeck*(deck, **"Tens row"**,13);  
  
  
Collections.*replaceAll*(deck, tensOfHearts, tensOfClubs);  
Card.*printDeck*(deck, **"Tens row"**,13);