Introduction to Collections

What is Java Collections?

Java Collections Framework

The Java platform includes a collections framework. A collection is an object that represents a group of objects (such as the classic Vector class). A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

The primary advantages of a collections framework are that it:

- **Reduces** programming effort by providing data structures and algorithms so you don't have to write them yourself.
- **Increases** performance by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- **Provides** interoperability between unrelated APIs by establishing a common language to pass collections back and forth.
- **Reduces** the effort required to learn APIs by requiring you to learn multiple ad hoc collection APIs.
- **Reduces** the effort required to design and implement APIs by not requiring you to produce ad hoc collections APIs.
- **Fosters** software reuse by providing a standard interface for collections and algorithms with which to manipulate them.

The collections framework consists of:

- **Collection interfaces**. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
- **General-purpose implementations.** Primary implementations of the collection interfaces.
- **Legacy implementations.** The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
- **Special-purpose implementations**. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations**. Implementations designed for highly concurrent use.
- **Wrapper implementations.** Add functionality, such as synchronization, to other implementations.
- **Convenience implementations.** High-performance "mini-implementations" of the collection interfaces.
- **Abstract implementations.** Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms.** Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure.** Interfaces that provide essential support for the collection interfaces.
- **Array Utilities.** Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

Collection Interfaces

What is Collection Interface?

A ***Collection*** represents a group of objects known as its elements. The ***Collection interface*** is used to pass around collections of objects where maximum generality is desired. For example, by convention all **general-purpose collection implementations** have a constructor that takes a **Collection argument**. This constructor, known as a **conversion constructor**, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to convert the collection's type.

The Most Common Collection Interfaces

List - A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.

Set - A Set is a Collection that cannot contain duplicate elements. There are three main implementations of Set interface: **HashSet, TreeSet,** and **LinkedHashSet.**

Queue - store elements FIFO (First In, First Out) - Like a stack

Map - key/value pairs

Iterator - access elements of collections

Before we get any further into Collection Interfaces and examples of them, we need to talk about & understand Class Wrappers

What is a Class Wrapper?

A **Wrapper class** is a class which contains the primitive data types (**int, char, short, byte, etc)**. In other words, **wrapper classes** provide a way to use primitive data types (int, char, short, byte, etc) as objects. These wrapper classes come under java.util package.

- boolean - Boolean
- char - Character
- byte - Byte
- short - Short
- int - Integer
- long - Long
- float - Float
- double - Double

Why do we need Class Wrappers?

- Wrapper Class will **convert primitive data types into objects.** The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are **passed by value)**.
- The **classes in java.util package** handles only objects and hence wrapper classes help in this case also.
- **Data structures** in the Collection framework such as **ArrayList** and **Vector** store only the objects (reference types) and not the **primitive types**.
- The object is needed to support **synchronization** in multithreading.

Let’s take a look at two very simple example code that displays the conversion.

Generics

What are Generics & Why do we use them?

In a nutshell, generics enable types (**classes and interfaces**) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

We use Generics because it implements...

- Stronger type checks at compile time.
  - A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
- Elimination of casting
- Enabling programmers to implement generic algorithms.
  - By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Using Generics example

without generics requires casting:

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

re-written to use generics, the code does not require casting:

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

Another Generic Example using ArrayList Collection Interface

```
import java.util.ArrayList;

class Main {
   public static void main(String[] args) {

      // create an array list to store Integer data
      ArrayList<Integer> list1 = new ArrayList<>();
      list1.add(4);
      list1.add(5);
      System.out.println("ArrayList of Integer: " + list1);

      // creates an array list to store String data
      ArrayList<String> list2 = new ArrayList<>();
      list2.add("Four");
      list2.add("Five");
      System.out.println("ArrayList of String: " + list2);

  }
}

// Output
ArrayList of Integer: [4, 5]
ArrayList of String: [Four, Five]
```

Okay, Back To Collection Interfaces

Methods of Collection Interface

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. The methods are following:

- add() - inserts the specified element to the collection
- size() - returns the size of the collection
- remove() - removes the specified element from the collection
- iterator() - returns an iterator to access elements of the collection
- addAll() - adds all the elements of a specified collection to the collection
- removeAll() - removes all the elements of the specified collection from the collection
- clear() - removes all the elements of the collection

Lets go over some method examples.. ***add()***

boolean addAll(Collection c, T... elements): This method adds all of the provided elements to the specified collection at once. The elements can be provided as a comma-separated list.

```
List fruits = new ArrayList();
Collections.addAll(fruits, "Apples", "Oranges",
"Banana");
fruits.forEach(System.out::println);
```

Output:

```
Apples
Oranges
Banana
```

size()...

The size() method of List interface in Java is used to get the number of elements in this list. That is, this method returns the count of elements present in this list container.

Syntax: public int size()

```
2
3  public class test {
4      public static void main(String[] args)
5      {
6          // Create an ArrayList of 4 colors
7            ArrayList<String> color_list = new ArrayList<String>(5
8          color_list.add(new String("While"));
9          color_list.add(new String("Black"));
10         color_list.add(new String("Red"));
11         color_list.add(new String("Green"));
12
13         // Get the size of the ArrayList.
14         int size = color_list.size();
15         System.out.println("ArrayList contains " + size +
16             " elements.");
17     }
18 }
```