Java SE & Spring

Module 1: Java SE

08.Java Collections Framework

What is Java Collections Framework?

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Collections are used in every programming language and when Java arrived, it also came with few Collection classes – Vector, Stack, Hashtable, Array.

The Java **Collections** Framework provides a set of interfaces and classes to implement various data structures and algorithms.

Benefits of Collections Framework

Java Collections framework have following benefits:

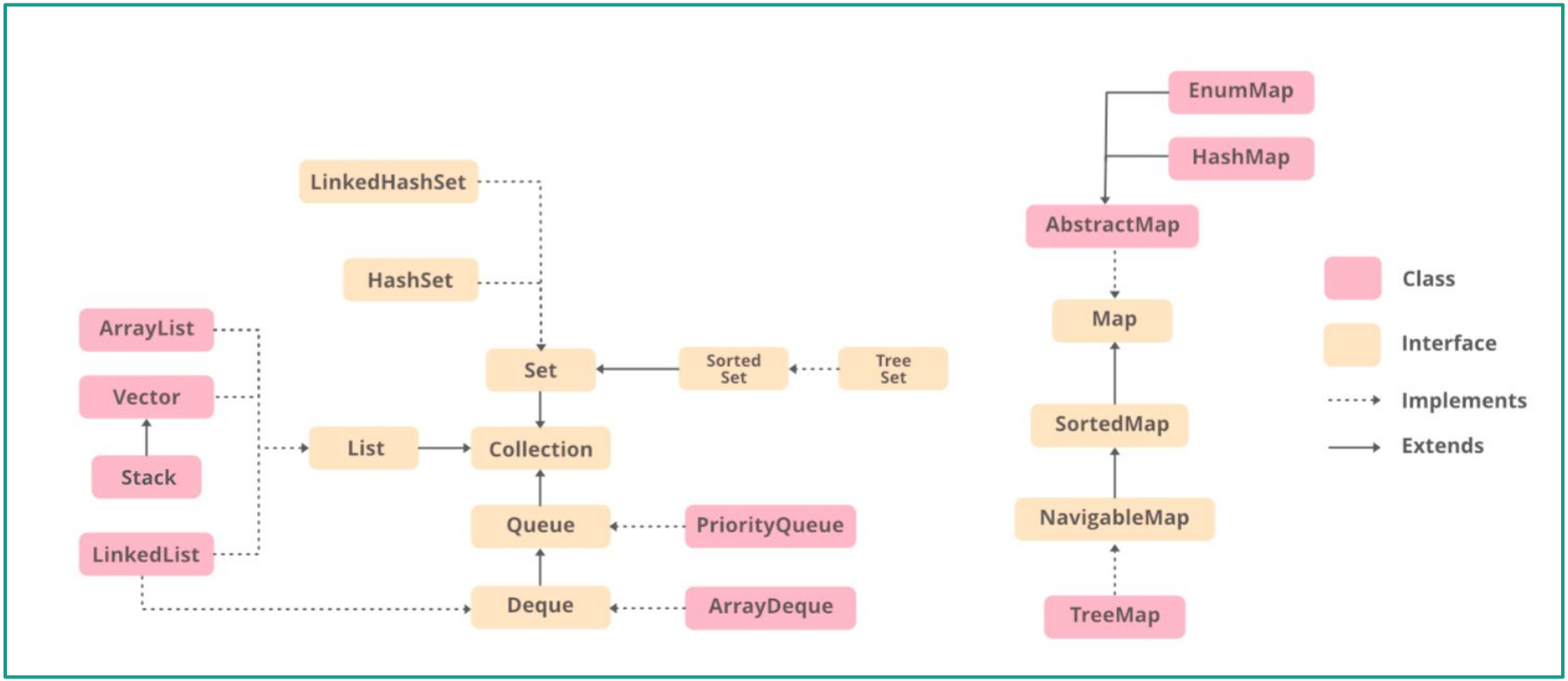
- **Reduced Development Effort:** It comes with almost all common types of collections and useful methods to iterate and manipulate the data. So we can concentrate more on business logic rather than designing our collection APIs.
- **Better Quality**: Using core collection classes that are well-tested increases our program quality rather than using any home-developed data structure.
- **Reduced effort to maintain** because everybody knows Collection API classes.

Java Collections API Interfaces

Java collection interfaces are the foundation of the Java Collections Framework. Note that all the core collection interfaces are generic; for example `public interface` Collection<E>. The **<E>** syntax is for Generics, which we will talk about it in the next chapter of the course. In this chapter, we will talk about five interfaces and a few most widely used implementation classes.:

1. **Collection** Interface
2. **List** Interface and ArrayList Class
3. **Set** Interface and HashSet Class
4. **Map** Interface and HashMap Class
5. **Iterator** Interface

Hierarchy of the Collection Framework



Collection Interface

The Collection interface is the root interface of the Java collections framework. There is no direct implementation of this interface. However, it is implemented through its subinterfaces like **List**, **Set**, and **Queue**.

For example, the **ArrayList** class implements the **List** interface which is a subinterface of the Collection Interface.

Methods of Collection Interface

The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its subinterfaces:

- **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

List Interface

In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the **Collection** interface. Since List is an interface, we cannot create objects from it. In order to use functionalities of the List interface, we can use these classes:

- **ArrayList**
- LinkedList
- Vector
- Stack

Methods of List Interface

The **List** interface includes all the methods of the **Collection** interface. Its because Collection is a super interface of List. Some of the commonly used methods of the Collection interface that's also available in the List interface are:

- **add()** - adds an element to a list
- **addAll()** - adds all elements of one list to another
- **get()** - helps to randomly access elements from lists
- **iterator()** - returns iterator object that can be used to sequentially access elements of lists
- **set()** - changes elements of lists

Methods of List Interface

- **remove()** - removes an element from the list
- **removeAll()** - removes all the elements from the list
- **clear()** - removes all the elements from the list (more efficient than removeAll())
- **size()** - returns the length of lists
- **toArray()** - converts a list into an array
- **contains()** - returns true if a list contains specified element

ArrayList Class

The **ArrayList** class of the Java collections framework provides the functionality of resizable-arrays.

It implements the List interface.

Syntax:

```
1  ArrayList<Type> arrayList =  new ArrayList<>();
2  // or
3  List<Type> arrayList =  new ArrayList<>();
```

We cannot use primitive types while creating an arraylist. Instead, we have to use the corresponding wrapper classes. Here, **Integer** is the corresponding wrapper class of **int**.

ArrayList Class

Example:

```
1  List<String> studentList =  new ArrayList<>();
2  studentList.add("Hasan");
3  studentList.add("Merve");
4  studentList.add("Osman");
5  studentList.add("Seda");
6
7  System.out.println(studentList.contains("Ali"));
8
9  System.out.println(studentList.get(0));
10 studentList.remove("Ali");
11 // studentList.clear();
12
13 for (String student : studentList) {
14     System.out.println(student);
15 }
```

ArrayList vs Array

In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it. To handle this issue, we can use the ArrayList class. It allows us to create resizable arrays. Unlike arrays, arraylists can automatically adjust its capacity when we add or remove elements from it. Hence, arraylists are also known as dynamic arrays.

.

Converting Array to ArrayList

We can also convert the array into an arraylist. For that, we use the asList() method of the Arrays class.

Example:

```
1  String[] languages = {"Java", "Javascript", "C++", "Python"};
2
3  for (String language : languages) {
4     System.out.println(language);
5  }
6
7  System.out.println("-----------------------------------");
8
9  List<String> languageList = Arrays.asList(languages);
10
11 for (String languageItem : languageList) {
12     System.out.println(languageItem);
13 }
```

Wrapper Classes

The wrapper classes in Java are used to convert primitive types (int, char, float, etc) into corresponding objects.

Each of the 8 primitive types has corresponding wrapper classes.

Primitive Data Type	Wrapper Class
`byte`	Byte
`short`	Short
`int`	Integer
`long`	Long
`float`	Float
`double`	Double
`char`	Character
`boolean`	Boolean

Converting Primitive Types to Wrapper Objects

Example:

```
1  int primitiveIntNumber = 5;
2  double primitiveDoubleNumber = 2.3;
3
4  // wrapping with using valueOf() method
5  Integer wrappedIntNumber1 = Integer.valueOf(primitiveIntNumber);
6  Double wrappedDoubleNumber1 = Double.valueOf(primitiveDoubleNumber);
7  System.out.println(wrappedIntNumber1);
8  System.out.println(wrappedDoubleNumber1);
9
10 //wrapping with directly assigning (this is called Autoboxing)
11 Integer wrappedIntNumber2 = primitiveIntNumber;
12 Double wrappedDoubleNumber2 = primitiveDoubleNumber;
13 System.out.println(wrappedIntNumber1);
14 System.out.println(wrappedDoubleNumber1);
```

Converting Wrapper Objects to Primitive Types

Example:

```
1  //converting wrapper objects to primitive types with ___Value() methods
2  int primitiveIntNumber2 = wrappedIntNumber1.intValue();
3  double primitiveDoubleNumber2 = wrappedDoubleNumber1.doubleValue();
4  System.out.println(primitiveIntNumber2);
5  System.out.println(primitiveDoubleNumber2);
6
7  Boolean wrapperBooleanValue = Boolean.TRUE;
8  boolean primitiveBooleanValue = wrapperBooleanValue.booleanValue();
9  System.out.println(wrapperBooleanValue);
10 System.out.println(primitiveBooleanValue);
11 //converting wrapper objects to primitive types with directly assigning (this is
12 //called unboxing)
13 boolean primitiveBooleanValue2 = wrapperBooleanValue;
14 primitiveIntNumber2 = wrappedIntNumber2;
15 primitiveDoubleNumber2 = wrappedDoubleNumber2;
16 System.out.println(primitiveIntNumber2);
17 System.out.println(primitiveDoubleNumber2);
```

Set Interface

The **Set** interface of the Java **Collections** framework provides the features of the mathematical set in Java. It extends the Collection interface. Unlike the **List** interface, sets cannot contain duplicate elements. Since Set is an interface, we cannot create objects from it. In order to use functionalities of the Set interface, we can use these classes:

- **HashSet**
- LinkedHashSet
- EnumSet
- TreeSet

These classes are defined in the **Collections** framework and implement the **Set** interface.

Methods of Set Interface

The **Set** interface includes all the methods of the **Collection** interface. It's because **Collection** is a super interface of **Set**. Some of the commonly used methods of the **Collection** interface that's also available in the **Set** interface are:

- **add()** - adds the specified element to the set
- **addAll()** - adds all the elements of the specified collection to the set
- **iterator()** - returns an iterator that can be used to access elements of the set sequentially
- **remove()** - removes the specified element from the set
- **removeAll()** - removes all the elements from the set that is present in another specified set
- **retainAll()** - retains all the elements in the set that are also present in another specified set

Methods of Set Interface

- **clear()** - removes all the elements from the set
- **size()** - returns the length (number of elements) of the set
- **toArray()** - returns an array containing all the elements of the set
- **contains()** - returns true if the set contains the specified element
- **containsAll()** - returns true if the set contains all the elements of the specified collection
- **hashCode()** - returns a hash code value (address of the element in the set)

HashSet Class

The **HashSet** class of the Java **Collections** framework provides the functionalities of the hash table data structure.

Syntax:

```
1  HashSet<Integer> numbers1 = new HashSet<>();
2  // or
3  Set<Integer> numbers1 = new HashSet<>();
```

As we discussed in ArrayList, we cannot use primitive types while creating an **HashSet**. Instead, we have to use the corresponding wrapper classes. Here, **Integer** is the corresponding wrapper class of **int**.

HashSet Class

Example:

```
1  Set<Integer> numberSet =  new HashSet<>();
2  numberSet.add(2);
3  numberSet.add(12);
4  numberSet.add(23);
5  numberSet.add(2);
6
7  System.out.println("Size of the numberSet: " + numberSet.size());
8
9  for (Integer number : numberSet) {
10     System.out.println(number);
11 }
12
13 System.out.println(numberSet);
```

Map Interface

The **Map** interface of the Java collections framework provides the functionality of the map data structure. In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual values. A map cannot contain duplicate keys. And, each key is associated with a single value.

Since Map is an interface, we cannot create objects from it.

In order to use functionalities of the Map interface, we can use these classes:

- **HashMap**
- EnumMap
- LinkedHashMap
- WeakHashMap
- TreeMap

Methods of Map Interface

The Map interface includes all the methods of the Collection interface. It is because Collection is a super interface of Map. Besides methods available in the Collection interface, the Map interface also includes the following methods:

- **put(K, V)** - Inserts the association of a key K and a value V into the map. If the key is already **present,** the new value replaces the old value.
- **putAll()** - Inserts all the entries from the specified map to this map.
- **putIfAbsent(K, V)** - Inserts the association if the key K is not already associated with the value V.
- **get(K)** - Returns the value associated with the specified key K. If the key is not found, it returns null.
- **getOrDefault(K, defaultValue)** - Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.

Methods of Map Interface

- **containsKey(K)** - Checks if the specified key K is present in the map or not.
- **containsValue(V)** - Checks if the specified value V is present in the map or not.
- **replace(K, V)** - Replace the value of the key K with the new specified value V.
- **replace(K, oldValue, newValue)** - Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.
- **remove(K)** - Removes the entry from the map represented by the key K.
- **remove(K, V)** - Removes the entry from the map that has key K associated with value V.
- **keySet()** - Returns a set of all the keys present in a map.
- **values()** - Returns a set of all the values present in a map.
- **entrySet()** - Returns a set of all the key/value mapping present in a map.

HashMap Class

The HashMap class of the Java collections framework provides the functionality of the hash table data structure.It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map. The HashMap class implements the Map interface.

Syntax:

```
1  HashMap<String, String> countryMap = new HashMap<>();
2  // or
3  Map<String, String> countryMap = new HashMap<>();
```

As we discussed in ArrayList and HashSet, we cannot use primitive types while creating an **HashMap**.

Instead, we have to use the corresponding wrapper classes.

HashMap Class

Example:

```
1  Map<String, String> countryMap =  new HashMap<>();
2  countryMap.put("TR", "TURKEY");
3  countryMap.put("AZ", "AZERBAIJAN");
4  countryMap.put("USA", "UNITED STATES");
5  countryMap.put("UK", "UNITED KINGDOM");
6  countryMap.put("TR2", "TURKEY");
7
8  for (String key : countryMap.keySet()) {
9      System.out.println(countryMap.get(key));
10 }
11
12 for (String value: countryMap.values()){
13     System.out.println(value);
14 }
```

Iterator Interface

The **Iterator** interface of the Java collections framework allows us to access elements of a collection. All the Java collections include an iterator() method. This method returns an instance of iterator used to iterate over elements of collections. The Iterator interface provides 4 methods that can be used to perform various operations on elements of collections.

- **hasNext()** - returns true if there exists an element in the collection
- **next()** - returns the next element of the collection
- **remove()** - removes the last element returned by the next()
- **forEachRemaining()** - performs the specified action for each remaining element of the collection

Iterator Interface

Example:

```
1  // Iterator example with ArrayList
2  Iterator<String> studentIterator = studentList.iterator();
3  while(studentIterator.hasNext()){
4       System.out.println(studentIterator.next());
5  }
6  // Iterator example with HashSet
7  Iterator<Integer> numSetIterator = numberSet.iterator();
8  while(numSetIterator.hasNext()){
9       System.out.println(numSetIterator.next());
10 }
```

Practice Time: Let’s Code Together!

Let’s code together!🚀

📌**Exercise**:

Suppose there is a school. We want to implement this school by using Collections framework. Suppose there are 3 classroom. Every class has a name and has students.

- Print out every class with its name and students individually.
- Add, update and remove new classrooms and students.

⛔**Restrictions**:

- There should be only one student with same student number.
- There should be only one classroom with a specific name.

You can choose best suitable Collection framework structure.

Questions ?

Next:Generics in Java