**Java Collections Framework Summary**

**1. Introduction to Collections Framework**

* The **Collections Framework** is a set of classes and interfaces for storing and processing data efficiently.
* Introduced in **Java SE 2 (1998)** and updated significantly in:
  + **Java SE 5**: Added Generics.
  + **Java 8**: Introduced Lambda expressions and Default methods.
* Provides various **data structures** to store, organize, and retrieve data efficiently.

**2. Key Components of Collections Framework**

**A. Main Interfaces**

* **Collection**: Root interface of the framework.
* **List**: Ordered collection allowing duplicates.
* **Set**: Unordered collection without duplicates.
* **Queue**: Stores elements in a specific order (FIFO/LIFO).
* **Map**: Stores key-value pairs.

**B. Implementations (Classes)**

| **Interface** | **Common Implementations** |
| --- | --- |
| **List** | ArrayList, LinkedList, Vector |
| **Set** | HashSet, TreeSet, LinkedHashSet |
| **Queue** | PriorityQueue, ArrayDeque |
| **Map** | HashMap, TreeMap, LinkedHashMap, ConcurrentHashMap |

**3. Choosing the Right Collection**

**Key Considerations:**

* **Iteration-based access** → Use List (ArrayList, LinkedList).
* **Key-value storage** → Use Map (HashMap, TreeMap).
* **No duplicate elements** → Use Set (HashSet, TreeSet).
* **FIFO order processing** → Use Queue (LinkedList, PriorityQueue).
* **Thread-safe collections** → Use ConcurrentHashMap, CopyOnWriteArrayList.

**4. Avoiding Legacy Classes**

* **Vector → Use ArrayList** (unless thread safety is needed).
* **Stack → Use ArrayDeque** (better performance).
* **Hashtable → Use HashMap or ConcurrentHashMap**.
* **Enumeration → Use Iterator**.

**5. Collections vs Arrays**

| **Feature** | **Arrays** | **Collections** |
| --- | --- | --- |
| Fixed Size | Yes | No (Resizable) |
| Allows Dynamic Operations | No | Yes (Add, Remove, Sort, etc.) |
| Allows Generics | No | Yes |
| Type-Safety | No | Yes (With Generics) |

**6. Examples of Common Collections Usage**

**List Example (ArrayList)**

```

import java.util.\*;

class Main {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

System.out.println(list); // Output: [Apple, Banana, Cherry]

}

}

```

**Set Example (HashSet)**

```

import java.util.\*;

class Main {

public static void main(String[] args) {

Set<Integer> set = new HashSet<>();

set.add(10);

set.add(20);

set.add(10); // Duplicate ignored

System.out.println(set); // Output: [10, 20]

}

}

```

**Map Example (HashMap)**

```

import java.util.\*;

class Main {

public static void main(String[] args) {

Map<String, Integer> map = new HashMap<>();

map.put("Apple", 3);

map.put("Banana", 2);

System.out.println(map.get("Apple")); // Output: 3

}

}

```

**Queue Example (PriorityQueue)**

```

import java.util.\*;

class Main {

public static void main(String[] args) {

Queue<Integer> pq = new PriorityQueue<>();

pq.add(5);

pq.add(1);

pq.add(3);

System.out.println(pq.poll()); // Output: 1 (smallest element)

}

}

```

**7. Iterating Over Collections**

**Using Iterator**

```

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

```

**Using For-Each Loop**

```

for (String item : list) {

System.out.println(item);

}

```

**8. Key Takeaways**

* The **Collections Framework** provides efficient data structures for storing and retrieving data.
* Choose the right **interface and implementation** based on the use case.
* Prefer **modern implementations** over legacy ones (ArrayList > Vector, HashMap > Hashtable).
* Use **iterators or enhanced for-loops** for better readability.

This summary will be useful for interview preparation and practical implementation in Java development.