Chapter 15: Collections and Generics

15.0 Introduction

In real-world applications, working with groups of objects is common—whether it's storing customer records, processing transactions, or managing a list of tasks. Java provides a robust **Collections Framework** to handle such tasks efficiently. Combined with **Generics**, which enable type-safe code and eliminate runtime errors caused by type casting, these features are indispensable for modern Java programming.

This chapter explores Java's **Collections Framework** and **Generics** in detail. You'll understand their architecture, how to use them effectively, and best practices for ensuring performance and type safety.

15.1 The Java Collections Framework Overview

15.1.1 What is a Collection?

A **Collection** is an object that groups multiple elements into a single unit. It is used to store, retrieve, manipulate, and communicate aggregate data.

15.1.2 Core Interfaces

- `Collection<E>`
- `List<E>`
- `Set<E>`
- `SortedSet<E>`
- `NavigableSet<E>`
- `Queue<E>`
- `Deque<E>`
- `Map<K, V>`
- `SortedMap<K, V>`
- `NavigableMap<K, V>`

Each interface defines operations and contracts for specific types of collections.

15.2 List Interface and Its Implementations

15.2.1 List Interface

A **List** is an ordered collection (also known as a sequence) that may contain duplicate elements.

15.2.2 Implementations

- **ArrayList**
  - Dynamic array-based.
  - Fast random access.
- **LinkedList**
  - Doubly-linked list.
  - Efficient insertions/deletions.
- **Vector**
  - Synchronized.
- **Stack**
  - LIFO stack built on Vector.

15.2.3 Key Methods

- `add(E e)`
- `remove(Object o)`
- `get(int index)`
- `set(int index, E element)`
- `iterator()`, `listIterator()`

15.3 Set Interface and Its Implementations

15.3.1 Set Interface

A **Set** is a collection that does not allow duplicate elements.

15.3.2 Implementations

- **HashSet**
  - Backed by a hash table.
  - Unordered.
- **LinkedHashSet**
  - Maintains insertion order.
- **TreeSet**
  - Sorted in natural or comparator order.

15.4 Queue and Deque

15.4.1 Queue Interface

Used for **FIFO** operations.

- `add()`, `remove()`, `peek()`, `poll()`

15.4.2 Deque Interface

Double-ended queue allowing **FIFO** and **LIFO**.

- `addFirst()`, `addLast()`, `removeFirst()`, `removeLast()`

15.4.3 Implementations

- `PriorityQueue` – for natural ordering or custom comparators.
- `ArrayDeque` – efficient resizable array-based implementation.

15.5 Map Interface and Its Implementations

15.5.1 Map Interface

A **Map** stores key-value pairs. Keys must be unique.

15.5.2 Implementations

- **HashMap**
  - Unordered, allows `null` key and values.
- **LinkedHashMap**
  - Maintains insertion order.
- **TreeMap**
  - Sorted by keys.
- **Hashtable**
  - Legacy synchronized implementation.

15.5.3 Common Methods

- `put(K key, V value)`
- `get(Object key)`
- `remove(Object key)`
- `keySet()`, `values()`, `entrySet()`

15.6 Iterating Over Collections

- **Enhanced For-Loop** (`for-each`)
- **Iterator** (with `hasNext()` and `next()`)
- **ListIterator** (for bidirectional access)
- **Streams and Lambdas** (Java 8+)

15.7 Algorithms in Collections

- `Collections.sort()`
- `Collections.reverse()`
- `Collections.shuffle()`
- `Collections.binarySearch()`
- `Collections.max()`, `min()`
- `Collections.unmodifiableList()`

15.8 Comparators and Comparable

15.8.1 Comparable Interface

- Defines natural ordering via `compareTo(T o)`.

15.8.2 Comparator Interface

- Defines custom ordering using `compare(T o1, T o2)`.

15.9 The Role of Generics

15.9.1 Why Generics?

- Type safety
- Elimination of casting
- Code reusability

15.9.2 Syntax

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

15.9.3 Generic Methods

```
public <T> void printArray(T[] array) {
    for (T item : array) System.out.println(item);
}
```

15.9.4 Bounded Type Parameters

```
<T extends Number>
```

15.10 Wildcards in Generics

15.10.1 Unbounded Wildcards: `<?>`

Used when the exact type is unknown.

15.10.2 Upper Bounded Wildcards: `<? extends T>`

Allows reading items of type `T` or its subtypes.

15.10.3 Lower Bounded Wildcards: `<? super T>`

Allows writing items of type `T` or its supertypes.

15.10.4 PECS Rule

- **Producer – extends**
- **Consumer – super**

15.11 Generic Classes and Interfaces

15.11.1 Generic Class

```
class Box<T> {
    private T value;
    public void set(T value) { this.value = value; }
    public T get() { return value; }
}
```

15.11.2 Generic Interface

```
interface DataStore<T> {
    void save(T item);
}
```

15.12 Collections vs Arrays

Feature	Collections	Arrays
Size	Dynamic	Fixed
Type Safety	With Generics	Compile-time only
Flexibility	High	Limited
Performance	Slight overhead	Faster for primitives

15.13 Best Practices

- Always use Generics to avoid `ClassCastException`.
- Prefer `List` over `ArrayList` in declarations.
- Use `Streams` for functional-style processing.
- Avoid raw types.
- Use appropriate collection type based on use case (e.g., `HashSet` for uniqueness).

15.14 Summary

The Java **Collections Framework** and **Generics** are cornerstones of effective Java development. Collections enable the management of groups of data, while Generics ensure that this management is both **type-safe** and **reusable**. Mastering these tools is crucial for building scalable, maintainable, and robust Java applications.
