Java Collections Framework

Great! Let's start learning the **Java Collections Framework** from scratch with detailed explanations and code examples.

📘 What is the Java Collections Framework (JCF)?

The **Java Collections Framework** is a set of classes and interfaces that implement commonly reusable data structures like:

- Lists
- Sets
- Queues
- Maps

It is found in the `java.util` package and helps manage **groups of objects**.

✅ Core Interfaces in the Java Collections Framework

1. Collection (root interface)

All collection interfaces like `List`, `Set`, and `Queue` extend `Collection`.

```java
Collection<String> collection = new ArrayList<>();
collection.add("Apple");
collection.add("Banana");
System.out.println(collection);
```

🔹 List Interface (Ordered Collection, Allows Duplicates)

Implementations:

- `ArrayList` (resizable array)
- `LinkedList` (doubly-linked list)
- `Vector` (synchronized, legacy)

➤ `ArrayList` Example

```java
import java.util.*;

public class ArrayListExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Banana");

        System.out.println("Fruits List: " + fruits);
        System.out.println("First fruit: " + fruits.get(0));
    }
}
```

◆ Set Interface (No Duplicates, Unordered or Sorted)

Implementations:

- **`HashSet`** (no order)
- **`LinkedHashSet`** (insertion order)
- **`TreeSet`** (sorted)

➤ `HashSet` Example

```java
import java.util.*;

public class HashSetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Mango");
        set.add("Apple");  // Duplicate, ignored

        System.out.println("Set: " + set);
    }
}
```

◆ Queue Interface (FIFO)

Implementations:

- **`LinkedList`** (common)
- **`PriorityQueue`** (natural order or custom)

➤ `Queue` Example

```java
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("A");
        queue.add("B");
        queue.add("C");
        System.out.println("Queue: " + queue);
        System.out.println("Removed: " + queue.poll()); // Removes "A"
        System.out.println("Queue after poll: " + queue);
    }
}
```

🔹 Map Interface (Key-Value Pairs, No Duplicate Keys)

Implementations:

- **`HashMap`** (unordered)
- **`LinkedHashMap`** (insertion order)
- **`TreeMap`** (sorted by keys)

➤ `HashMap` Example

```java
import java.util.*;

public class HashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(101, "John");
        map.put(102, "Alice");
        map.put(101, "Bob"); // Replaces value for key 101

        System.out.println("Map: " + map);
        System.out.println("Value for 102: " + map.get(102));
    }
}
```

🔄 Iterating Over Collections

➤ `for-each` loop

```java
List<String> list = Arrays.asList("A", "B", "C");
for (String item : list) {
    System.out.println(item);
}
```

➤ Using Iterator

```java
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

🛠️ Utility Class: `Collections`

```java
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 5, 30, 25);
        Collections.sort(numbers);  // Ascending
        System.out.println("Sorted: " + numbers);

        Collections.reverse(numbers);
        System.out.println("Reversed: " + numbers);

        Collections.shuffle(numbers);
        System.out.println("Shuffled: " + numbers);
    }
}
```

🔐 Thread-safe Variants

Use **`Collections.synchronizedList()`** for thread-safe collections:

```java
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
```

📌 Summary Table

Interface	Implementation	Ordered	Allows Duplicates	Sorted
List	ArrayList, LinkedList	Yes	Yes	No
Set	HashSet, TreeSet	No	No	TreeSet only
Queue	LinkedList, PriorityQueue	Yes	Yes	PriorityQueue (by priority)
Map	HashMap, TreeMap	N/A	Keys No, Values Yes	TreeMap (by key)

Would you like:

- Practice assignments after each topic?
- A mini-project using collections?
- More advanced topics like custom sorting, streams, or concurrent collections?

Let me know how you'd like to proceed!

Great! Let's dive into `Iterator` and `ListIterator` in Java with clear explanations and examples.

🔁 `Iterator` vs `ListIterator` in Java

Feature	`Iterator`	`ListIterator`
Applicable to	All `Collection` types	Only `List` implementations
Direction	Forward only	Forward and backward
Can remove elements?	Yes	Yes
Can add elements?	❌ No	✅ Yes
Can replace elements?	❌ No	✅ Yes (`set()`)

✅ `Iterator` – Traverse Any Collection (Forward Only)

➤ Syntax and Example

```java
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

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

        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
    }
}
```

➤ `iterator.remove()` Example

```java
import java.util.*;

public class IteratorRemoveExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(Arrays.asList("Alice", "Bob", "Charlie"));

        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            String name = iterator.next();
            if (name.equals("Bob")) {
                iterator.remove(); // safely removes from list
            }
        }

        System.out.println("Updated List: " + names);
    }
}
```

⚠️ Never use `list.remove(item)` inside the loop when using `iterator`. It throws `ConcurrentModificationException`.

🔄 `ListIterator` – Only for Lists, Supports Bi-directional Traversal

➤ Forward and Backward Traversal

```
java

import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> cities = new ArrayList<>(Arrays.asList("Delhi", "Mumbai", "Chennai"));

        ListIterator<String> listIterator = cities.listIterator();

        System.out.println("Forward direction:");
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }

        System.out.println("Backward direction:");
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

➤ Modifying Elements with `ListIterator`

```
java

import java.util.*;

public class ListIteratorModifyExample {
    public static void main(String[] args) {
        List<String> items = new ArrayList<>(Arrays.asList("Pen", "Pencil", "Eraser"));

        ListIterator<String> listIterator = items.listIterator();

        while (listIterator.hasNext()) {
            String item = listIterator.next();
            if (item.equals("Pencil")) {
                listIterator.set("Marker");  // Replaces Pencil with Marker
            }
        }

        System.out.println("Updated List: " + items);
    }
}
```

➤ Adding Elements with `ListIterator`

```
java

import java.util.*;

public class ListIteratorAddExample {
    public static void main(String[] args) {
        List<String> animals = new ArrayList<>(Arrays.asList("Cat", "Dog"));

        ListIterator<String> listIterator = animals.listIterator();

        while (listIterator.hasNext()) {
            String animal = listIterator.next();
            if (animal.equals("Cat")) {
                listIterator.add("Tiger"); // Adds after "Cat"
            }
        }

        System.out.println("Updated List: " + animals);
    }
}
```

🧠 Quick Summary

Method in Iterator / ListIterator	Description
`hasNext()`	Checks if next element exists
`next()`	Moves to and returns next element
`remove()`	Removes last element returned
`hasPrevious()` *(ListIterator)*	Checks if previous element exists
`previous()` *(ListIterator)*	Returns the previous element
`add(E e)` *(ListIterator)*	Inserts element
`set(E e)` *(ListIterator)*	Replaces last returned element

Would you like:

- An assignment to practice **`Iterator`** and **`ListIterator`**?
- How they're used in real-world projects?
- Differences with Java 8+ forEach and streams?

Let me know how you'd like to continue!

Great question! Let's explore how `Iterator` and `ListIterator` are used in **real-world Java projects**, especially in enterprise applications like e-commerce, HRM, inventory, etc.

💼 Real-World Use Cases of `Iterator` and `ListIterator`

---

1. Filtering Data (e.g., Removing Inactive Users)

Scenario: In an HRM system, you want to clean up a list of employee records and remove users who are marked inactive.

```java
List<Employee> employees = employeeService.getAllEmployees();
Iterator<Employee> iterator = employees.iterator();

while (iterator.hasNext()) {
    Employee emp = iterator.next();
    if (!emp.isActive()) {
        iterator.remove();  // safely remove inactive employees
    }
}
```

> ✅ Why `Iterator`? Because it safely removes elements during traversal.

---

2. Bi-directional Editing (e.g., Inventory Adjustments)

Scenario: In an inventory system, you want to traverse a list of products forward and backward to adjust names or prices.

```java
List<Product> products = productService.getProducts();
ListIterator<Product> listIterator = products.listIterator();

while (listIterator.hasNext()) {
    Product p = listIterator.next();
    if (p.getName().equals("OldProduct")) {
        listIterator.set(new Product("UpdatedProduct", p.getPrice()));
    }
}

while (listIterator.hasPrevious()) {
    Product p = listIterator.previous();
    System.out.println("Revisiting: " + p.getName());
}
```

> ✅ Why `ListIterator`? Because it allows updates and backward navigation.

---

3. Custom Business Rules (e.g., Validating Orders)

Scenario: In an e-commerce platform, before processing orders, you validate if products are in stock.

java

```
List<OrderItem> items = order.getItems();
Iterator<OrderItem> iterator = items.iterator();

while (iterator.hasNext()) {
    OrderItem item = iterator.next();
    if (!inventoryService.isInStock(item.getProductId())) {
        iterator.remove();  // Remove out-of-stock items from order
    }
}
```

---

4. Rollback Scenarios (e.g., Transaction Reversal)

Scenario: You need to reverse the order of transactions (e.g., undo feature in a text editor or rollback in banking software).

java

```
List<Transaction> history = transactionService.getHistory();
ListIterator<Transaction> listIterator = history.listIterator(history.size());  // Start from the end

while (listIterator.hasPrevious()) {
    Transaction txn = listIterator.previous();
    txn.rollback();
}
```

> ✅ Why **`ListIterator`** with **`hasPrevious()`**? Perfect for reverse iteration.

---

5. Modifying List During Workflow Execution

Scenario: In workflow engines, you may want to add or remove steps dynamically.

java

```
List<String> steps = new ArrayList<>(List.of("Login", "Verify", "Pay"));

ListIterator<String> iterator = steps.listIterator();

while (iterator.hasNext()) {
    String step = iterator.next();
    if (step.equals("Verify")) {
        iterator.add("Scan");  // Add a new step after "Verify"
    }
}
System.out.println(steps);
```

💡 Why Use Iterators Instead of Index-Based Loops?

- Prevent **`ConcurrentModificationException`**
- Cleaner and more readable
- Work with all Collection types (not just Lists)
- Allow safe mutation (remove, add, update) during iteration

🔧 Alternative in Modern Java (Streams)

In modern Java (Java 8+), **`Stream API`** is often used **instead of `Iterator`** when mutation is not required:

```java
List<String> names = List.of("John", "Alex", "Bob");

names.stream()
     .filter(name -> !name.equals("Bob"))
     .forEach(System.out::println);
```

> But when mutation/removal during traversal is needed → **`Iterator`** and **`ListIterator`** are still preferred.

✅ Summary

Use Case	Use	Preferred Tool
Filter and remove items	Safely remove while looping	**`Iterator`**
Modify list during traversal	Add, update, or reverse loop	**`ListIterator`**
Read-only filtering or mapping	Transform without modifying	**`Stream API`**
Reverse traversal	Backward movement	**`ListIterator`**

Would you like:

- A **mini-project** that uses **`Iterator`** and **`ListIterator`**?
- **Assignments** to practice real-world scenarios?
- Or continue to the **Java Collections Algorithms** (like sort, shuffle, rotate)?

Let me know!