Advanced Programming (OOP)

Module 5: Collection Framework

SDB

Spring 2025

Module 5: Topics

- Introduction to Java Collection Framework.
- Common Interfaces and Classes: List, Set, Map.
- Iterators and Streams.
- Sorting and Comparing Objects.
- Practical Examples and Use Cases.

Outline

Iterable Interface

- The **Iterable** interface is the root interface for all collection classes.
- It represents a collection that can be traversed using an iterator.
- Provides a single method: `iterator()`.

Example Code:

```
import java.util.*;
class IterableExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("A", "B", "C");
        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Iterator in Java

Definition: The Iterator interface provides a way to traverse elements in a collection sequentially without exposing the underlying implementation.

Key Methods:

- **hasNext()** - Returns true if there are more elements to iterate.
- **next()** - Returns the next element in the iteration.
- **remove()** - Removes the last element returned by the iterator.
- **forEachRemaining(Consumer**$\langle ?\ super\ E \rangle$ **action)** - Performs an action on remaining elements.

Use Cases:

- Used when modifying a collection during iteration.
- Prevents ConcurrentModificationException when iterating.
- Supports forward and bidirectional traversal (‘ListIterator‘).

Example: Using Iterator

Iterating Over a List Using Iterator:

```java
import java.util.*;

public class IteratorExample {
    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(); // Safe removal
            }
        }
        System.out.println(names);
    }
}
```

Iterator vs Iterable in Java

Iterable Interface:

- Represents a collection that can be iterated using an ‘Iterator‘.
- Defines a single abstract method: ‘iterator()‘.
- Enables ‘for-each‘ loop functionality.

Iterator Interface:

- Provides methods to traverse and manipulate elements in a collection.
- Defines methods: ‘hasNext()‘, ‘next()‘, ‘remove()‘, and ‘forEachRemaining()‘.
- Used explicitly when modifying a collection during iteration.

Key Differences:

Feature	Iterable	Iterator
Definition	Represents a collection	Used to traverse a collection
Method	‘iterator()‘	‘hasNext()‘, ‘next()‘, ‘remove()‘
Usage	Supports ‘for-each‘ loop	Requires explicit calls
Modification	Cannot modify collection	Can remove elements safely

The Enhanced For-Loop

Definition: The enhanced ‘for-each‘ loop is a more readable alternative to iterating collections using traditional iterators or indexed loops.

Key Features:

- Provides a simplified syntax for iterating over collections and arrays.
- Eliminates the need for manually handling ‘Iterator‘ objects.
- Ensures type safety by leveraging generics.

Usage of Enhanced For-Loop with Collections

Applicable Collections:

- ‘List$\langle E \rangle$‘ - Iterates through ordered elements.
- ‘Set$\langle E \rangle$‘ - Iterates through unique elements.
- ‘Queue$\langle E \rangle$‘ - Iterates through queue elements in order.

Benefits:

- Reduces boilerplate code compared to traditional ‘for‘ loops.
- Prevents ‘ConcurrentModificationException‘ by avoiding manual iterator modifications.
- Works seamlessly with Java’s ‘Iterable‘ interface.

Comparison: Enhanced For-Loop I

Enhanced For-Loop:

- Shorter, more readable syntax.
- Cannot modify collection while iterating.
- Best suited for read-only operations.

Iterator:

- Allows modification ('remove()' method) while iterating.
- Useful for concurrent collections and custom iteration logic.
- Provides fine-grained control over element retrieval.

Comparison: Enhanced For-Loop II

```java
// Example of enhanced for-loop iterating over a list
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

for (String name : names) {
    System.out.println(name);
}

// Example of enhanced for-loop iterating over a set
Set<Integer> numbers = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));

for (int num : numbers) {
    System.out.println(num);
}

// Example of enhanced for-loop iterating over an array
int[] values = {10, 20, 30, 40};

for (int value : values) {
    System.out.println(value);
}
```

Java Streams API

Definition: Streams provide functional-style operations for processing sequences of elements in a declarative way.

Key Features:

- Supports operations like filtering, mapping, reducing, and sorting.
- Works with collections, arrays, and I/O channels.
- Lazy evaluation improves performance.
- Supports parallel execution with parallel streams.

Main Functions Related to Collections:

- **filter()** - Filters elements based on a condition.
- **map()** - Transforms each element.
- **sorted()** - Sorts the stream elements.
- **forEach()** - Iterates through elements.
- **collect()** - Converts stream to collection.

Example: Using Streams in Java

Filtering and Sorting a List:

```
import java.util.*;
import java.util.stream.*;

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

        List<String> filteredNames = names.stream()
            .filter(name -> name.startsWith("A")) // Filter names starting with "A"
            .sorted() // Sort alphabetically
            .collect(Collectors.toList()); // Collect as a List

        System.out.println(filteredNames);
    }
}
```

Outline

What is the Java Collection Framework?

Definition: A unified architecture for storing and manipulating groups of objects.

Key Benefits:

- Reduces programming effort.
- Increases performance through high-performance implementations.
- Provides algorithms for sorting and searching.

Common Interfaces:

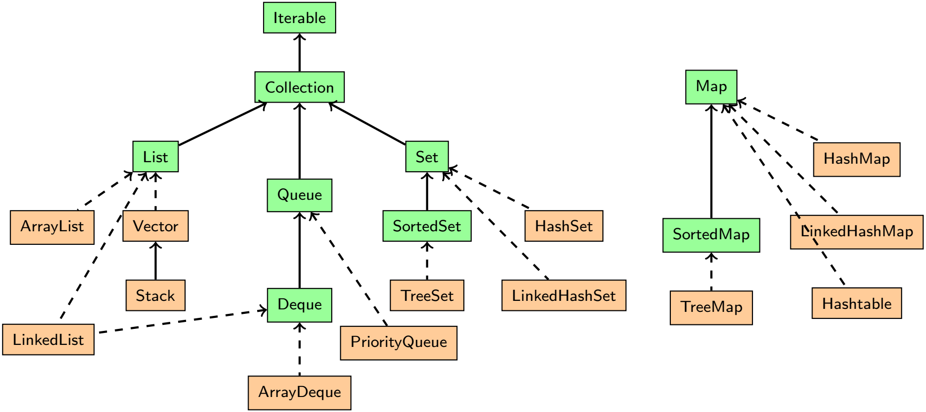
- **List:** Ordered collection, allows duplicates.
- **Set:** Unordered collection, no duplicates.
- **Map:** Key-value pairs, no duplicate keys.

Java Collections Overview

What Collections Framework Provides:

- **Interfaces:**
  - Collection
  - List
  - Set
  - Queue
  - Map
- **Implementations:**
  - ArrayList, LinkedList (List)
  - HashSet, LinkedHashSet, TreeSet (Set)
  - PriorityQueue, Deque (Queue)
  - HashMap, LinkedHashMap, TreeMap (Map)
- **Utility Classes:**
  - Collections
  - Arrays
- **Features:**
  - Sorting
  - Searching
  - Iteration
  - Thread-safe collections (e.g., ConcurrentHashMap)

Java Collections Framework Hierarchy



Collection Interface

- The **Collection** interface extends `Iterable` and is the root interface for Java collections.
- Defines common methods like `add()`, `remove()`, and `size()`.

Example Code:

```java
import java.util.*;
class CollectionExample {
    public static void main(String[] args) {
        Collection<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("Size: " + numbers.size());
    }
}
```

Outline

List Interface

- Defines an ordered collection (sequence)
- Allows duplicate elements
- Supports positional access and iteration
- Common implementations: ArrayList, LinkedList, Vector, Stack

Important Methods:

- add(E e), get(int index), remove(int index), size()

Usage Scenario: Managing ordered data collections, like to-do lists.

```java
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list.get(0)); // Apple
```

Abstract List Class

- Provides a skeletal implementation of the List interface
- Reduces the effort required to implement a List
- Implements some methods while leaving others abstract

Important Methods:

- add(E e), remove(int index), iterator()

Usage Scenario: When creating custom list implementations.

```
abstract class MyList<E> extends AbstractList<E> {
    public boolean add(E e) {
        // Custom add implementation
        return true;
    }
}
```

Abstract Sequential List Class

- Extends AbstractList
- Designed for sequential access data structures
- Used by LinkedList

Important Methods:

- listIterator(), get(int index)

Usage Scenario: Optimized for sequential access rather than random access.

ArrayList Class

- Implements a dynamic array
- Provides fast random access (O(1) time complexity)
- Slower insertions and deletions compared to LinkedList
- Resizable, with automatic capacity expansion

Important Methods:

- add(E e), get(int index), remove(int index)

Usage Scenario: When frequent retrieval of elements is required.

```java
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(10);
numbers.add(20);
System.out.println(numbers.get(1)); // 20
```

Vector Class

- Similar to ArrayList but synchronized
- Thread-safe, but has performance overhead
- Rarely used in modern Java applications

Important Methods:

- add(E e), remove(int index), size()

Usage Scenario: When working in a multi-threaded environment.

```
Vector<String> vector = new Vector<>();
vector.add("Thread-safe");
System.out.println(vector.get(0));
```

Stack Class

- Extends Vector
- Implements a LIFO (Last In, First Out) stack
- Provides push, pop, peek, empty, and search methods

Important Methods:

- push(E e), pop(), peek()

Usage Scenario: Implementing undo functionality or expression evaluation.

```java
Stack<Integer> stack = new Stack<>();
stack.push(5);
stack.push(10);
System.out.println(stack.pop()); // 10
```

LinkedList Class

- Implements a doubly-linked list
- Efficient insertions and deletions (O(1) for adding/removing at ends)
- Slower random access compared to ArrayList (O(n) time complexity)
- Implements both List and Deque interfaces

Important Methods:

- addFirst(E e), addLast(E e), removeFirst(), removeLast()

Usage Scenario: When frequent insertions and deletions are required.

```java
LinkedList<String> queue = new LinkedList<>();
queue.add("First");
queue.add("Second");
System.out.println(queue.removeFirst()); // First
```

Outline

Sorting in Java Collections

Definition: Sorting is the process of arranging elements in a specific order, typically ascending or descending.

Key Methods:

- **Collections.sort(List**$\langle T \rangle$ **list)** - Sorts a list in natural order.
- **Collections.sort(List**$\langle T \rangle$ **list, Comparator**$\langle ?\ \mathit{super}\ \ T \rangle$ **c)** - Sorts using a custom comparator.
- **Arrays.sort(T[] array)** - Sorts an array in natural order.
- **Arrays.sort(T[] array, Comparator**$\langle ?\ \mathit{super}\ \ T \rangle$ **c)** - Sorts an array using a comparator.

Sorting Order:

- Uses Comparable (natural ordering) or Comparator (custom ordering).
- Uses Timsort (optimized merge and insertion sort) for Lists.

Example: Sorting in Java

Sorting a List of Strings:

```java
import java.util.*;

public class SortingExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Charlie", "Alice", "Bob");
        Collections.sort(names); // Natural order sorting
        System.out.println(names);
    }
}
```

Sorting a List of Integers in Descending Order:

```java
List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
Collections.sort(numbers, Collections.reverseOrder());
System.out.println(numbers);
```

Sorting Using Comparable and Comparator I

Comparable Interface:

- Natural ordering.
- Implemented in the class to be sorted.

Comparator Interface:

- Custom ordering.
- Implemented as a separate class.

Example: Comparator

```
import java.util.*;

class Student {
    String name;
    int marks;

    Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    @Override
    public String toString() {
```

Sorting Using Comparable and Comparator II

```java
        return name + " - " + marks;
    }
}

class MarksComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return Integer.compare(s1.marks, s2.marks);
    }
}

public class Test {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student("Alice", 85));
        students.add(new Student("Bob", 92));
        students.add(new Student("Charlie", 78));

        Collections.sort(students, new MarksComparator());
        for (Student s : students) {
            System.out.println(s);
        }
    }
}
```

Listing 1: Custom Sorting Using Comparator

Searching in Java Collections

Definition: Searching is the process of finding an element in a collection.
Key Methods:

- **Collections.binarySearch(List$\langle T \rangle$ list, T key)** - Searches for an element in a sorted list.
- **Arrays.binarySearch(T[] array, T key)** - Searches for an element in a sorted array.
- **contains(Object o)** - Checks if a collection contains an element.
- **indexOf(Object o)** - Finds the index of an element in a List.

Conditions:

- Binary search requires the list or array to be sorted.
- ‘contains()‘ and ‘indexOf()‘ have O(n) complexity, while ‘binarySearch()‘ has O(log n).

Example: Searching in Java

Using Binary Search on a Sorted List:

```
import java.util.*;
public class SearchingExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
        int index = Collections.binarySearch(numbers, 30);
        System.out.println("Element found at index: " + index);
    }
}
```

Checking if an Element Exists in a Collection:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
boolean exists = names.contains("Bob");
System.out.println("Is Bob in the list? " + exists);
```

Outline

Map Interface

- Stores key-value pairs
- No duplicate keys allowed

Important Methods:

- put(K key, V value), get(Object key), remove(Object key)

Usage Scenario: Storing mappings of unique keys to values.

```
Map<String, Integer> map = new HashMap<>();
map.put("A", 1);
map.put("B", 2);
System.out.println(map.get("A")); // 1
```

Different Types of Map Implementations I

- **SortedMap Interface**
  - Extends Map, maintains sorted key order
  - Implemented by TreeMap
- **NavigableMap Interface**
  - Extends SortedMap with navigation methods
  - Implemented by TreeMap and ConcurrentSkipListMap
- **ConcurrentMap Interface**
  - A thread-safe Map interface
  - Implemented by ConcurrentHashMap
- **TreeMap Class**
  - Implements SortedMap using a Red-Black tree
  - Maintains natural/comparator order

Usage Scenario: Maintaining a sorted mapping of keys.

Different Types of Map Implementations II

- **AbstractMap Class**
  - A skeletal implementation of the Map interface

  Usage Scenario: Simplifies the creation of custom map implementations.
- **ConcurrentHashMap Class**
  - A thread-safe, high-performance map
  - Supports concurrent reads/writes

  Usage Scenario: Multi-threaded applications requiring efficient maps.
- **EnumMap Class**
  - A high-performance map specialized for enum keys

  Usage Scenario: Mapping values to a fixed set of enum keys.
- **HashMap Class**
  - Implements Map using a hash table
  - Allows null keys and values

  Usage Scenario: General-purpose key-value storage.

Different Types of Map Implementations III

- **IdentityHashMap Class**
  - Uses reference equality instead of value equality

  Usage Scenario: When key identity matters over content equality.
- **LinkedHashMap Class**
  - Maintains insertion order of entries

  Usage Scenario: Caching with predictable iteration order.
- **HashTable Class**
  - A synchronized implementation of Map
  - Legacy class, replaced by ConcurrentHashMap

  Usage Scenario: Older applications requiring thread safety.
- **Properties Class**
  - Extends HashTable, used for configuration settings

  Usage Scenario: Storing and managing application properties.

Outline

Queue Interface

- Defines a collection for holding elements prior to processing
- Typically follows FIFO (First-In-First-Out) order
- Common implementations: LinkedList, PriorityQueue, ArrayDeque

Important Methods:

- add(E e), offer(E e), poll(), peek()

Usage Scenario: Task scheduling, buffering requests.

```
Queue<String> queue = new LinkedList<>();
queue.add("Task1");
queue.add("Task2");
System.out.println(queue.poll()); // Task1
```

Different Implementation of Queue I

- **BlockingQueue Interface**
  - Extends Queue interface with blocking operations
  - Useful in concurrent programming
  - Implementations: ArrayBlockingQueue, LinkedBlockingQueue

  Important Methods:
  - put(E e), take()

  Usage Scenario: Producer-consumer problems.

```
BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);
queue.put(1);
System.out.println(queue.take());
```

- **AbstractQueue Class**
  - Provides a skeletal implementation of the Queue interface
  - Reduces the effort of implementing custom queues

  Usage Scenario: When creating custom queue implementations.

Different Implementation of Queue II

- **PriorityQueue Class**
  - Implements a priority heap-based queue
  - Elements are ordered based on natural ordering or a comparator

  Important Methods:
  - offer(E e), poll(), peek()

  Usage Scenario: Task scheduling with priorities.

```java
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(10);
pq.add(5);
System.out.println(pq.poll()); // 5
```

- **PriorityBlockingQueue Class**
  - A thread-safe PriorityQueue
  - Does not block on insertion

Different Implementation of Queue III

- **ConcurrentLinkedQueue Class**
  - A non-blocking, thread-safe queue
  - Uses CAS operations for thread safety

  Usage Scenario: Multi-threaded environments requiring fast, lock-free queues.
- **ArrayBlockingQueue Class**
  - A bounded blocking queue backed by an array
- **DelayQueue Class**
  - Holds elements until a delay expires

  Usage Scenario: Task scheduling where execution is delayed.
- **LinkedBlockingQueue Class**
  - A blocking queue backed by a linked list
- **LinkedTransferQueue Class**
  - A transfer queue allowing direct hand-off between producers and consumers

Deque Interface

- Double-ended queue, allows insertion and removal from both ends
- Implementations: ArrayDeque, LinkedList

Important Methods:

- addFirst(E e), addLast(E e), removeFirst(), removeLast()

Usage Scenario: Implementing stacks and queues efficiently.

```java
Deque<Integer> deque = new ArrayDeque<>();
deque.addFirst(10);
deque.addLast(20);
System.out.println(deque.removeFirst()); // 10
```

Different Deque Implementations I

- **BlockingDeque Interface**
  - Extends Deque with blocking operations
  - Used in multi-threaded applications
- **ConcurrentLinkedDeque Class**
  - A thread-safe, lock-free deque implementation

  Usage Scenario: Concurrent programming requiring double-ended queue access.
- **ArrayDeque Class**
  - A resizable-array implementation of Deque
  - Faster than Stack and LinkedList for stack operations

  Usage Scenario: Implementing a stack with better performance.

```java
ArrayDeque<String> stack = new ArrayDeque<>();
stack.push("Item1");
stack.push("Item2");
System.out.println(stack.pop()); // Item2
```

Outline

Set Interface

- A collection that does not allow duplicate elements
- Common implementations: HashSet, LinkedHashSet, TreeSet

Important Methods:

- add(E e), remove(Object o), contains(Object o)

Usage Scenario: Storing unique elements, eliminating duplicates.

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("B");
System.out.println(set.contains("A")); // true
```

Different Types of Set Implementations I

- **AbstractSet Class**
  - A skeletal implementation of the Set interface
  - Reduces the effort needed to implement custom sets

  Usage Scenario: When creating custom set implementations.
- **CopyOnWriteArraySet Class**
  - A thread-safe Set implementation
  - Uses CopyOnWriteArrayList internally

  Usage Scenario: When iteration needs to be thread-safe without locking.
- **EnumSet Class**
  - A specialized Set implementation for enum types
  - Very efficient in terms of performance

  Usage Scenario: Handling a fixed set of constants efficiently.
- **ConcurrentHashMap Class**
  - A thread-safe, high-performance hash table

Different Types of Set Implementations II

- Supports concurrent reads and writes

Usage Scenario: Multi-threaded applications requiring high-performance maps.

- **HashSet Class**
  - Implements Set using a HashMap internally
  - Provides constant-time performance for basic operations

  Usage Scenario: Maintaining unique elements without ordering.

```java
Set<Integer> hashSet = new HashSet<>();
hashSet.add(1);
hashSet.add(2);
System.out.println(hashSet.contains(1)); // true
```

Different Types of Set Implementations III

- **LinkedHashSet Class**
  - Maintains insertion order
  - Extends HashSet with a linked list

  Usage Scenario: Storing unique elements while maintaining insertion order.
- **SortedSet Interface**
  - Extends Set to provide ordering of elements
  - Implemented by TreeSet

  Important Methods:
  - first(), last(), headSet(E toElement), tailSet(E fromElement)
- **NavigableSet Interface**
  - Extends SortedSet with navigation methods
  - Implemented by TreeSet and ConcurrentSkipListSet

  Important Methods:
  - lower(E e), floor(E e), ceiling(E e), higher(E e)

Different Types of Set Implementations IV

- **TreeSet Class**
  - Implements a Red-Black tree-based set
  - Maintains elements in natural or comparator order

Usage Scenario: Storing sorted unique elements.

```
TreeSet<Integer> treeSet = new TreeSet<>();
treeSet.add(3);
treeSet.add(1);
treeSet.add(2);
System.out.println(treeSet); // [1, 2, 3]
```

- **ConcurrentSkipListSet Class**
  - A thread-safe, scalable sorted set
  - Implements NavigableSet and uses a skip list

Usage Scenario: Concurrent applications requiring a sorted set.

Outline

Miscellaneous I

How to convert HashMap to ArrayList

- Convert keys or values to an ArrayList.

```
Map<String, Integer> map = new HashMap<>();
map.put("A", 1);
map.put("B", 2);
List<String> keys = new ArrayList<>(map.keySet());
List<Integer> values = new ArrayList<>(map.values());
```

Randomly select items from a List

- Use Random class to get a random item.

```
List<String> list = Arrays.asList("A", "B", "C", "D");
Random rand = new Random();
String randomItem = list.get(rand.nextInt(list.size()));
```

Miscellaneous II

How to add all items from a collection to an ArrayList

- Use addAll() method.

```
List<Integer> list1 = new ArrayList<>(Arrays.asList(1, 2, 3));
List<Integer> list2 = new ArrayList<>(Arrays.asList(4, 5, 6));
list1.addAll(list2);
```

Conversion of Java Maps to List

- Convert entries to a List.

```
Map<String, Integer> map = new HashMap<>();
map.put("A", 1);
map.put("B", 2);
List<Map.Entry<String, Integer>> entries = new ArrayList<>(map.entrySet());
```

Array to ArrayList Conversion

- Use Arrays.asList() or new ArrayList¡¿(Arrays.asList()).

```
String[] array = {"A", "B", "C"};
List<String> list = new ArrayList<>(Arrays.asList(array));
```

ArrayList to Array Conversion

- Use toArray() method.

```
List<String> list = Arrays.asList("A", "B", "C");
String[] array = list.toArray(new String[0]);
```

Differences between Array and ArrayList

- Array has a fixed size, while ArrayList is dynamic.
- Array can hold both primitive and object types, while ArrayList can only hold objects.
- ArrayList provides more built-in methods for manipulation.

Outline

Introduction to Generics

Definition: Generics in Java allow type parameters to be used in class, interface, and method definitions, enabling strong type checking and code reusability.

Key Features:

- Provides compile-time type safety, reducing ‘ClassCastException‘ at runtime.
- Allows code reuse without compromising type safety.
- Supports parameterized types, ensuring flexibility while maintaining strict type constraints.
- Eliminates the need for explicit type casting, enhancing readability.

Generic Classes and Methods

Generic Class Syntax:

- Defined using angle brackets '$\langle T \rangle$' to represent a generic type parameter.
- Allows creating a class that can work with different data types.

Generic Method Syntax:

- Defined using '$\langle T \rangle$' before the return type.
- Enables defining a method that can operate on different types while maintaining type safety.

Bounded Type Parameters

Definition: A bounded type parameter restricts the types that can be used as arguments.
Syntax: '$\langle T\ extends\ Number\rangle$' restricts T to 'Number' or its subclasses (e.g., 'Integer', 'Double').
Use Cases:

- Ensuring mathematical operations only work with numeric types.
- Restricting type parameters for collections requiring specific constraints.

Wildcard Parameters in Generics

Definition: Wildcards ('?') allow flexibility in defining generic types while ensuring compatibility with unknown types.

Types of Wildcards:

- '?' - Represents an unknown type.
- '? extends T' - Allows any type that is a subclass of 'T'.
- '? super T' - Allows any type that is a superclass of 'T'.

Use Cases:

- Reading elements from a collection while maintaining type safety.
- Writing elements to a collection with type constraints.

Generics in Collections

Benefits of Generics in Collections:

- Provides type safety when working with lists, sets, and maps.
- Eliminates the need for explicit casting.
- Enhances readability and maintainability of code.

Example Implementations:

- 'List$\langle String \rangle$ names = new ArrayList$\langle\rangle$();' - Ensures only 'String' objects are stored.
- 'Map$\langle Integer, String \rangle$ idToName = new HashMap$\langle\rangle$();' - Maps integers to string values safely.

Outline

Exercises for Students

1. **Using List:**
   - Create a ‘ShoppingList‘ program using ‘ArrayList‘.
   - Add, remove, and display items in the list.
2. **Using Set:**
   - Write a program to store unique employee IDs using ‘HashSet‘.
   - Display the IDs in sorted order using ‘TreeSet‘.
3. **Using Map:**
   - Create a ‘StudentMarks‘ program using ‘HashMap‘ to store student names and marks.
   - Allow searching for a student by name.
4. **Sorting:**
   - Implement sorting of a custom ‘Product‘ class by price using both ‘Comparable‘ and ‘Comparator‘.

Exercises: Choose the Right Collection I

1. **Scenario:** Store a list of student names with duplicates allowed.
   Hint: Use a List implementation.
2. **Scenario:** Maintain a unique set of employee IDs with no specific order.
   Hint: Use a Set implementation.
3. **Scenario:** Keep a sorted collection of product prices.
   Hint: Use TreeSet.
4. **Scenario:** Map employee IDs to their names for quick lookup.
   Hint: Use a Map implementation.
5. **Scenario:** Implement a task queue where tasks are processed in order of priority.
   Hint: Use PriorityQueue.
6. **Scenario:** Create a collection to store books in a library system, allowing fast insertion and retrieval based on book IDs.
   Hint: Use HashMap.

Exercises: Choose the Right Collection II

7. **Scenario:** Process a large dataset with frequent concurrent reads and updates.
   Hint: Use ConcurrentHashMap.
8. **Advanced Scenario:** Design a messaging system where messages need to be processed in the order they are received but should allow duplicate messages.
   Hint: Use LinkedList.
9. **Advanced Scenario:** Create a leaderboard system that maintains scores in descending order with no duplicate entries.
   Hint: Use TreeSet with a custom comparator.
10. **Advanced Scenario:** Implement a caching mechanism where the least recently accessed items are removed when the cache limit is reached.
    Hint: Use LinkedHashMap with access order.

Exercises: Choose the Right Collection III

11. **Advanced Scenario:** Build an inventory system that supports grouping items by category, allowing fast lookups for items in a category.
    Hint: Use a Map where keys are categories and values are Lists of items.
12. **Advanced Scenario:** Manage flight bookings such that the same seat cannot be booked twice and bookings should be sorted by seat numbers.
    Hint: Use TreeSet.
13. **Advanced Scenario:** Implement a task scheduler where tasks are prioritized based on urgency and must be executed in that order.
    Hint: Use PriorityQueue with a custom comparator.
14. **Advanced Scenario:** Develop a voting system that counts votes for candidates, ensuring real-time updates and efficient lookups.
    Hint: Use ConcurrentHashMap.

Advanced Scenarios: Combining Multiple Collections I

Scenario 1: E-commerce Order Management

- Store a mapping of order IDs to their details using **HashMap**.
- Maintain a sorted list of order timestamps using **TreeSet** for tracking recent orders.

Scenario 2: Social Media Platform

- Use a **HashMap** to map user IDs to their profile details.
- Store the list of followers for each user using a **HashSet** to avoid duplicates.

Scenario 3: Library Management System

- Maintain a catalog of books categorized by genre using a **HashMap** (genre as key, list of books as value).
- Track issued books using a **LinkedHashMap** for maintaining insertion order.

Advanced Scenarios: Combining Multiple Collections II

Scenario 4: Online Examination System

- Use a **TreeMap** to store question IDs sorted by difficulty.
- Maintain a **PriorityQueue** for real-time grading based on submission time.

Scenario 5: Movie Ticket Booking System

- Use a **HashMap** to store show timings as keys and available seats as values (List of seat numbers).
- Use a **TreeSet** to ensure seat numbers are stored in sorted order.

Scenario 6: Task Assignment System

- Use a **HashMap** to assign employees to tasks (employee ID as key, list of tasks as value).
- Maintain a **PriorityQueue** for tasks sorted by priority.