CORE JAVA

JAVA COLLECTIONS

1

INTRODUCTION TO JAVA COLLECTIONS

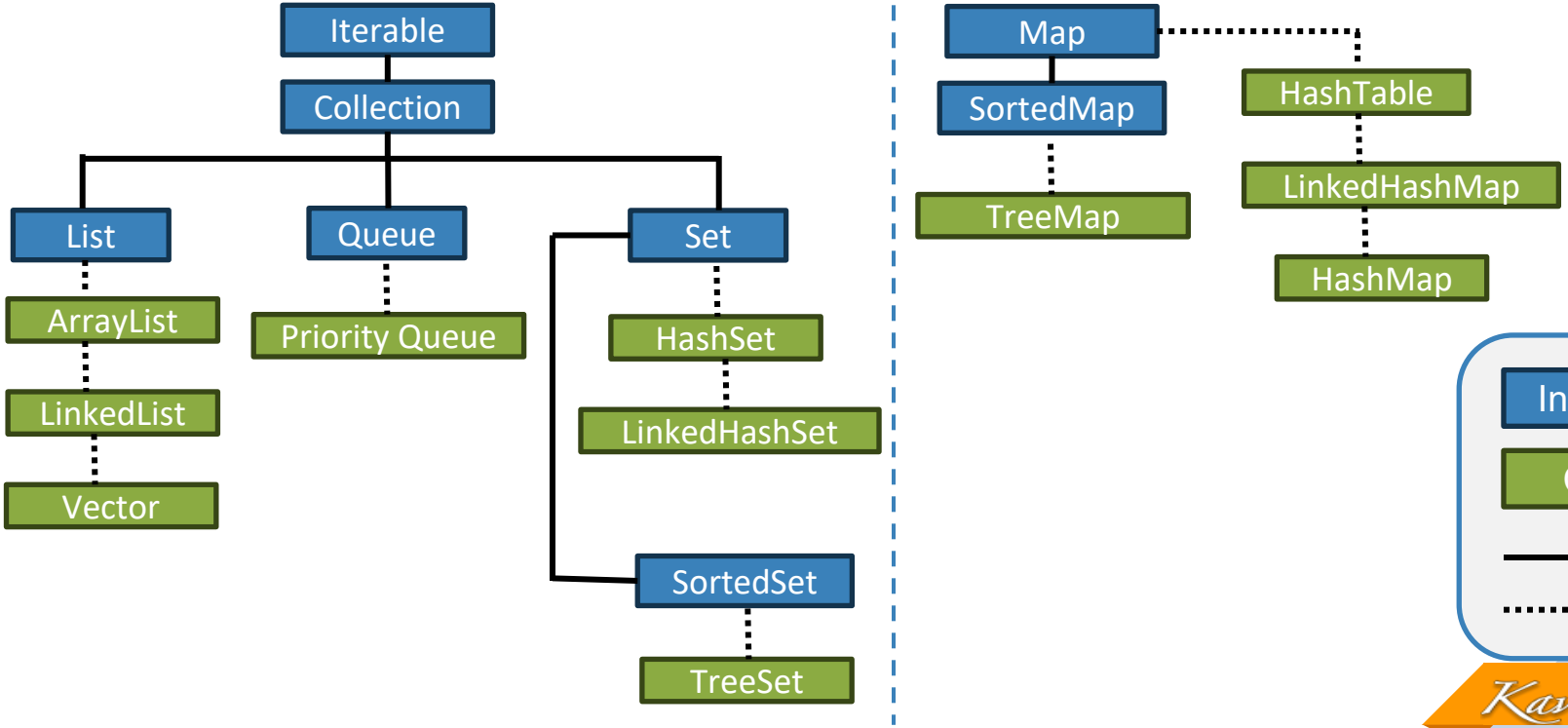
1

INTRODUCTION TO JAVA COLLECTIONS

Learning Objectives

- Understand the purpose of the java collections framework
- Collection interface hierarchy
- What is iterable interface?
- What is collection?
- Key features of a collection
- Key interfaces in collections
- Benefits of using collections
- Core interfaces of java collections

COLLECTION INTERFACE HIERARCHY



WHAT IS ITERABLE INTERFACE?

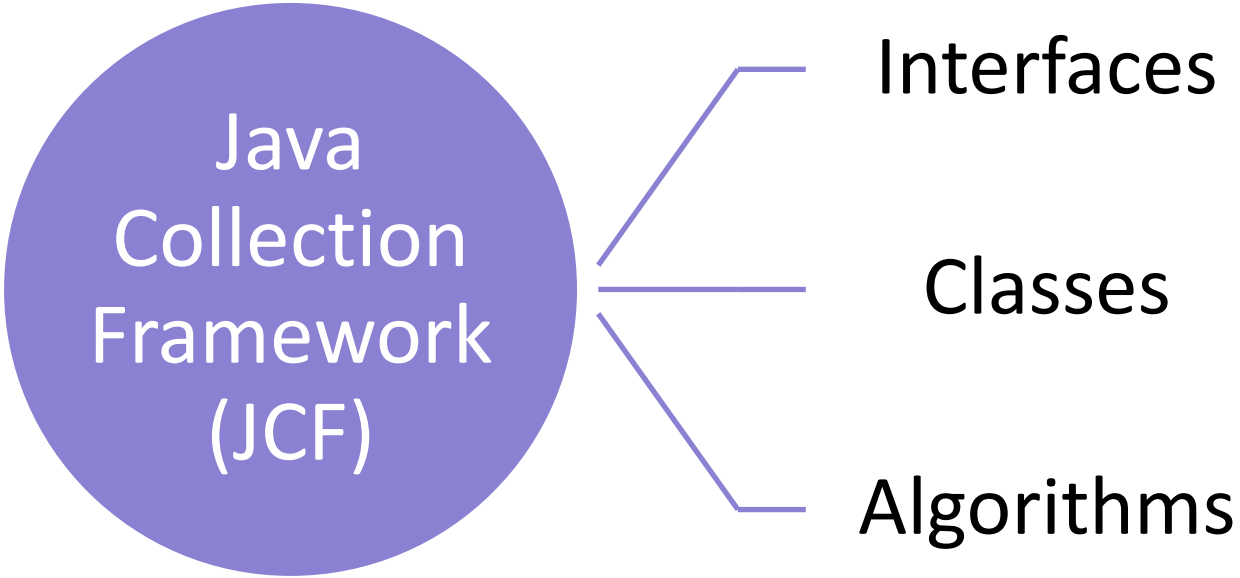
- The Iterable interface is the root interface of the Java Collections Framework, which represents a collection of objects that can be iterated over one by one

- It is part of the **java.lang** package, meaning any class that implements Iterable can be used with enhanced for-loops (for-each loop)

WHAT IS COLLECTION?

- A Collection is a framework that provides an architecture to store and manipulate a group of objects
- It serves as the root interface for the Java Collections Framework (JCF), which includes multiple interfaces, classes, and algorithms to manage collections of objects efficiently
- Collections are used to store, retrieve, manipulate, and communicate aggregate data

WHAT IS COLLECTION?



KEY FEATURES OF A COLLECTION

Group of Objects

A collection is essentially a container for holding multiple objects, often referred to as elements or items.

Interfaces and Classes

It provides interfaces like List, Set, Queue, and Map, and corresponding classes that implement them, such as ArrayList, HashSet, and HashMap.

Dynamic Nature

Collections are dynamic, meaning that the size of a collection can change as elements are added or removed.

Flexible Storage

Collections store references to objects. It can store objects of any type and handle heterogeneity.

Utility Methods

The Collections framework provides various methods for operations like searching, sorting, shuffling, reversing, and more.

KEY INTERFACES IN COLLECTIONS

Collection Interface

This is the root interface for all other collection interfaces like List, Set, and Queue.

List Interface

Represents an ordered collection (like an array) but with dynamic size.

Set Interface

Represents a collection that does not allow duplicates.

Queue Interface

Represents a collection designed for holding elements prior to processing, typically in a FIFO (First-In-First-Out) order.

Map Interface

Not a part of the Collection hierarchy but still part of the Collections Framework. It stores data in key-value pairs.

BENEFITS OF USING COLLECTIONS

Ease of Use

- Provides predefined data structures like lists, sets, and maps, reducing the need for custom implementations.

Efficiency

- Built-in optimization for storing, retrieving, and manipulating large data sets.

Generic

- Collections use generics, allowing for type safety and flexibility in storing data of various types.

Thread-Safety

- Some collection classes provide thread-safe implementations, like ConcurrentHashMap, ensuring safe access to data in multi-threaded environments.

CORE INTERFACES OF JAVA COLLECTIONS

List

- Ordered collection that allows duplicates
- Ideal for use cases like transaction histories where the order of elements matters and duplicates are expected
- Common Implementations: **ArrayList, LinkedList**

Set

- Unordered collection that does not allow duplicates
- Best suited for scenarios like unique customer IDs where duplicates must be avoided
- Common Implementations: **HashSet, LinkedHashSet, TreeSet**

Map

- Key-value pair storage. Each key maps to a value, and keys are unique
- Perfect for mapping account numbers to customer information
- Common Implementations: **HashMap, LinkedHashMap, TreeMap**

2

LIST

Learning Objectives

- What is list interface?
- Introduction to the list interface
- Arraylist key points & Example code
- Linkedlist key points & example code
- Vector key points & example code
- Comparing arraylist, linkedlist, and vector

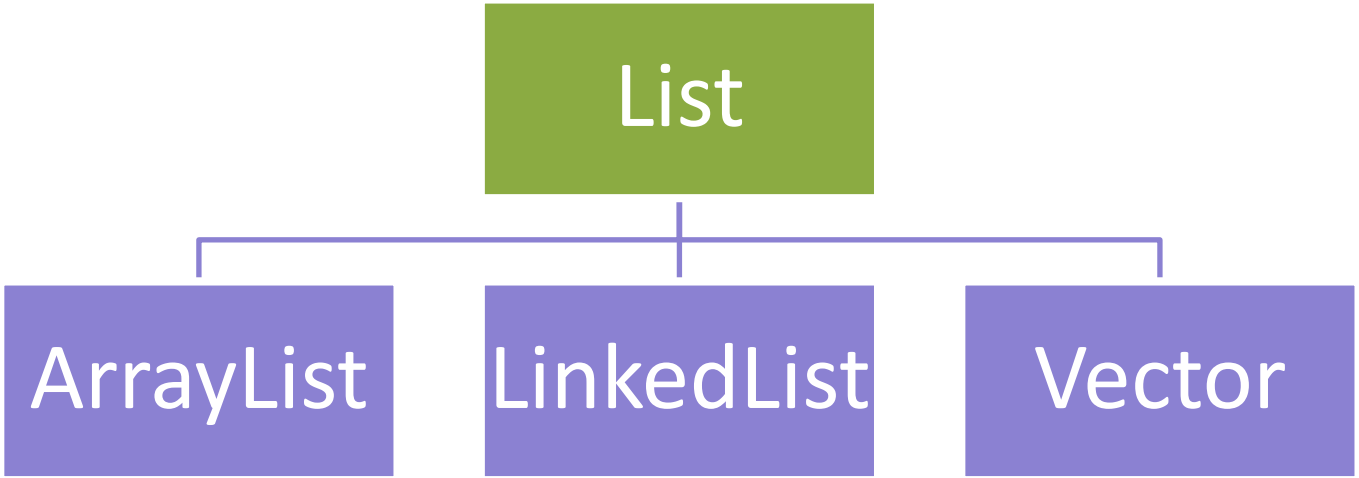
WHAT IS LIST INTERFACE?

The List interface represents an ordered collection of elements.

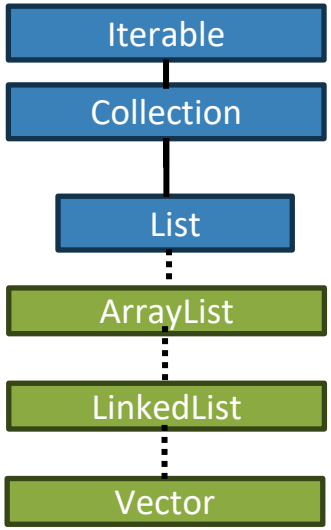
Lists allow indexed access, meaning each element can be accessed based on its position.

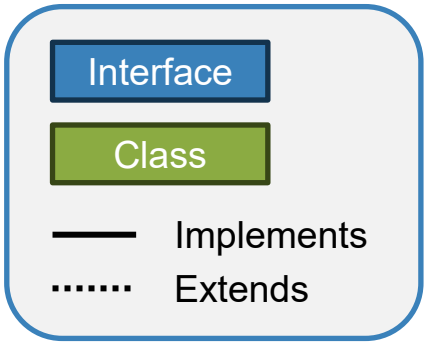
Duplicates are allowed, making it useful when maintaining a collection where repeated entries are expected (e.g., transaction histories in banking).

INTRODUCTION TO THE LIST INTERFACE



INTRODUCTION TO THE LIST INTERFACE





ARRAYLIST KEY POINTS

- A resizable array implementation of the List interface.
- Provides constant-time random access (O(1)), making it fast for retrieving elements by index.
- Best suited for use cases where retrieval is frequent, but insertions/deletions are less common.
- Frequently Used Methods in **ArrayList**
  - **add**(E e) – Adds an element to the list.
  - **get**(int index) – Retrieves the element at the specified index.
  - **remove**(int index) – Removes the element at the specified index.
  - **size**() – Returns the number of elements in the list.
  - **contains**(Object o) – Checks if the list contains the specified element.
  - **clear**() – Removes all elements from the list.

ARRAYLIST EXAMPLE

```java
import java.util.ArrayList;

public class BankAccount {
    public static void main(String[] args) {
        ArrayList<String> transactions = new ArrayList<>();
        transactions.add("Deposit: $500");
        transactions.add("Withdrawal: $200");
        transactions.add("Deposit: $300");
        for (String transaction : transactions) {
            System.out.println(transaction);
        }
        System.out.println("First transaction: " + transactions.get(0));
    }
}
```

```
Deposit: $500
Withdrawal: $200
Deposit: $300
First transaction: Deposit: $500
```

LINKEDLIST KEY POINTS

- A doubly linked list implementation of the List interface.
- Elements are stored in nodes where each node points to the next and previous element.
- Best suited for use cases where frequent insertions or deletions at the beginning or middle of the list are needed.
- Frequently Used Methods in **LinkedList**
  - **add()**: Adds an element to the end of the list.
  - **addFirst() / addLast()**: Adds an element to the beginning or end of the list.
  - **remove()**: Removes the first occurrence of an element.
  - **removeFirst() / removeLast()**: Removes the first or last element.
  - **get()**: Retrieves an element at a specific position.
  - **getFirst() / getLast()**: Retrieves the first or last element.

LINKEDLIST EXAMPLE

```java
import java.util.LinkedList;

public class InsuranceClaims {
    public static void main(String[] args) {
        LinkedList<String> claims = new LinkedList<>();

        claims.add("Claim 001: Car Accident");
        claims.add("Claim 002: Home Fire");
        claims.addFirst("Claim 000: Health Emergency");

        String firstClaim = claims.removeFirst();
        System.out.println("Processing: " + firstClaim);

        System.out.println("Pending claims: " + claims);
    }
}
```

```
Processing: Claim 000: Health Emergency
Pending claims: [Claim 001: Car Accident, Claim
002: Home Fire]
```

VECTOR KEY POINTS

- Synchronized version of the ArrayList, making it thread-safe by default.
- Like ArrayList, Vector stores elements in a resizable array but ensures thread safety through synchronization.
- Best suited for multi-threaded environments where collections are accessed by multiple threads concurrently.
- Frequently Used Methods in Vector
  - **add()**: Adds an element to the end of the vector.
  - **remove()**: Removes an element at a specific position or the first occurrence.
  - **get()**: Retrieves an element at a specific index.
  - **size()**: Returns the number of elements in the vector.
  - **isEmpty()**: Checks if the vector is empty.
  - **capacity()**: Returns the current capacity of the vector.
  - **trimToSize()**: Trims the vector capacity to match the size.

VECTOR EXAMPLE

```java
import java.util.Vector;

public class InsurancePolicies {
    public static void main(String[] args) {
        Vector<String> policies = new Vector<>();

        policies.add("Policy 001: Life Insurance");
        policies.add("Policy 002: Auto Insurance");
        policies.add("Policy 003: Health Insurance");

        for (String policy : policies) {
            System.out.println(policy);
        }

        policies.remove(1);
        System.out.println("Remaining policies: " + policies);
    }
}
```

```
Policy 001: Life Insurance
Policy 002: Auto Insurance
Policy 003: Health Insurance
Remaining policies: [Policy 001: Life Insurance,
Policy 003: Health Insurance]
```

COMPARING ARRAYLIST, LINKEDLIST, AND VECTOR

Feature	ArrayList	LinkedList	Vector
Data Structure	Resizable array	Doubly linked list	Resizable array
Access Speed	Fast (O(1))	Slow (O(n))	Fast (O(1))
Insertion/Deletion	Slow (O(n))	Fast at head/tail (O(1))	Slow (O(n))
Thread Safety	Not synchronized	Not synchronized	Synchronized
Best Use Case	Frequent access, fewer modifications	Frequent modifications	Multi-threaded access

3

SET

Learning Objectives

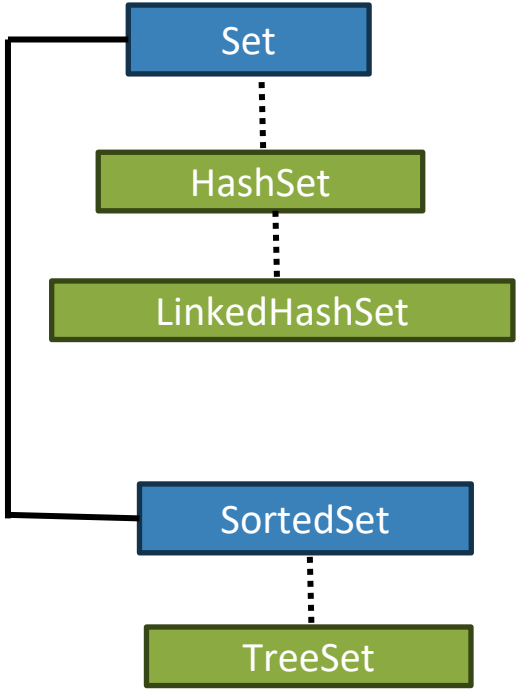
- What is **set** interface?
- Introduction to the set interface
- HashSet key points & Example code
- LinkedHashSet key points & example code
- TreeSet key points & example code
- Comparing HashSet, LinkedHashSet, and TreeSet

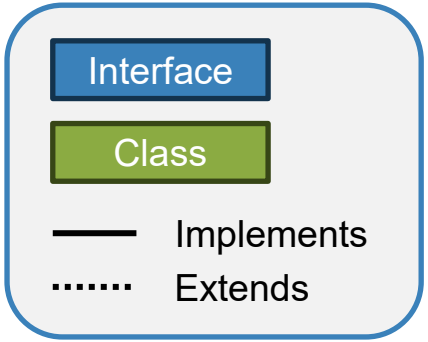
WHAT IS SET INTERFACE?

The Set interface represents a collection of unique elements; it does not allow duplicates.

Sets are typically used when you need to store a collection of items where order is not important, but uniqueness is.

INTRODUCTION TO THE SET INTERFACE





HASHSET KEY POINTS

- Implements the Set interface and is backed by a hash table.
- Provides constant-time performance for basic operations like add(), remove(), and contains() (O(1)), assuming a well-distributed hash function.
- Best suited for storing large collections where uniqueness is critical, but ordering is not important.
- Frequently Used Methods in HashSet
  - **add**(E e): Adds an element to the set.
  - **contains**(Object o): Checks if the set contains a specific element.
  - **remove**(Object o): Removes a specific element from the set.
  - **size()**: Returns the number of elements in the set.
  - **isEmpty()**: Checks if the set is empty.
  - **clear():** Removes all elements from the set.
  - **iterator():** Returns an iterator over the elements in the set.

HASHSET EXAMPLE

```java
import java.util.HashSet;
import java.util.Set;

public class Bank {
    public static void main(String[] args) {
        Set<String> customerIds = new HashSet<>();

        customerIds.add("CUST001");
        customerIds.add("CUST002");
        customerIds.add("CUST003");

        customerIds.add("CUST002");
        System.out.println("Customer IDs: " + customerIds);

        if (customerIds.contains("CUST001")) {
            System.out.println("Customer CUST001 exists");
        }

        customerIds.remove("CUST003");
        System.out.println("Updated Customer IDs: " + customerIds);
    }
}
```

```
Customer IDs: [CUST003, CUST002, CUST001]
Customer CUST001 exists
Updated Customer IDs: [CUST002, CUST001]
```

LINKEDHASHSET KEY POINTS

- Implements the Set interface and maintains a linked list of entries to preserve the insertion order.
- Provides the same performance as HashSet for basic operations, but with the added benefit of maintaining order.
- Best suited when you need to maintain a collection of unique elements but also want to preserve the order in which they were inserted.
- Frequently Used Methods in LinkedHashSet
  - **add**(E e): Adds an element while preserving insertion order.
  - **remove**(Object o): Removes an element from the set.
  - **contains**(Object o): Checks for an element.
  - **size**(): Returns the number of elements in the set.
  - **isEmpty**(): Checks if the set is empty.
  - **clear**(): Removes all elements.
  - **iterator**(): Returns an iterator over the set elements in insertion order.

LINKEDHASHSET EXAMPLE

```java
 import java.util.LinkedHashSet;
import java.util.Set;

public class BankLinkedHashSet {
   public static void main(String[] args) {
      Set<String> transactions = new LinkedHashSet<>();

      transactions.add("Deposit: $500");
      transactions.add("Withdraw: $200");
      transactions.add("Deposit: $300");

      System.out.println("Transaction History: " + transactions);

      transactions.add("Deposit: $500");
      System.out.println("After Attempting Duplicate: " + transactions);

      if (transactions.contains("Withdraw: $200")) {
         System.out.println("Withdraw: $200 exists in history");
      }
   }
}
```

```
Transaction History: [Deposit: $500, Withdraw: $200,
Deposit: $300]
After Attempting Duplicate: [Deposit: $500, Withdraw:
$200, Deposit: $300]
Withdraw: $200 exists in history
```

TREESET KEY POINTS

- Implements the NavigableSet interface (a subtype of Set), backed by a Red-Black Tree.
- Provides logarithmic time performance (O(log n)) for basic operations like add(), remove(), and contains().
- Best suited when you need a collection of unique elements, and the elements must be sorted (e.g., natural ordering or using a custom comparator).
- Frequently Used Methods in TreeSet
  - **add**(E e): Adds an element while maintaining sorted order.
  - **first**(): Returns the first (lowest) element.
  - **last**(): Returns the last (highest) element.
  - **remove**(Object o): Removes a specific element.
  - **contains**(Object o): Checks if the set contains the element.
  - **size**(): Returns the size of the set.
  - **iterator**(): Returns an iterator over the elements in ascending order.

TREESET EXAMPLE

```java
import java.util.TreeSet;
import java.util.Set;

public class BankTreeSet {
    public static void main(String[] args) {
        Set<Integer> accountBalances = new TreeSet<>();

        accountBalances.add(2500);
        accountBalances.add(1000);
        accountBalances.add(5000);

        System.out.println("Sorted Account Balances: " + accountBalances);

        System.out.println("Lowest Balance: " + ((TreeSet<Integer>) accountBalances).first());
        System.out.println("Highest Balance: " + ((TreeSet<Integer>) accountBalances).last());

        accountBalances.remove(1000);
        System.out.println("Updated Balances: " + accountBalances);
    }
}
```

```
Sorted Account Balances: [1000, 2500, 5000]
Lowest Balance: 1000
Highest Balance: 5000
Updated Balances: [2500, 5000]
```

COMPARING HASHSET, LINKEDHASHSET, AND TREESET

Feature	HashSet	LinkedHashSet	TreeSet
Data Structure	Hash table	Hash table + Linked List	Red-Black Tree
Search Performance	Fast (O(1))	Fast (O(1))	Moderate (O(log n))
Insertion/Deletion	Fast (O(1))	Fast (O(1))	Moderate (O(log n))
Ordering	None	Maintains insertion order	Sorted (natural or custom)
Best Use Case	Uniqueness with no order	Uniqueness with insertion order	Uniqueness with sorting

4

MAP

Learning Objectives

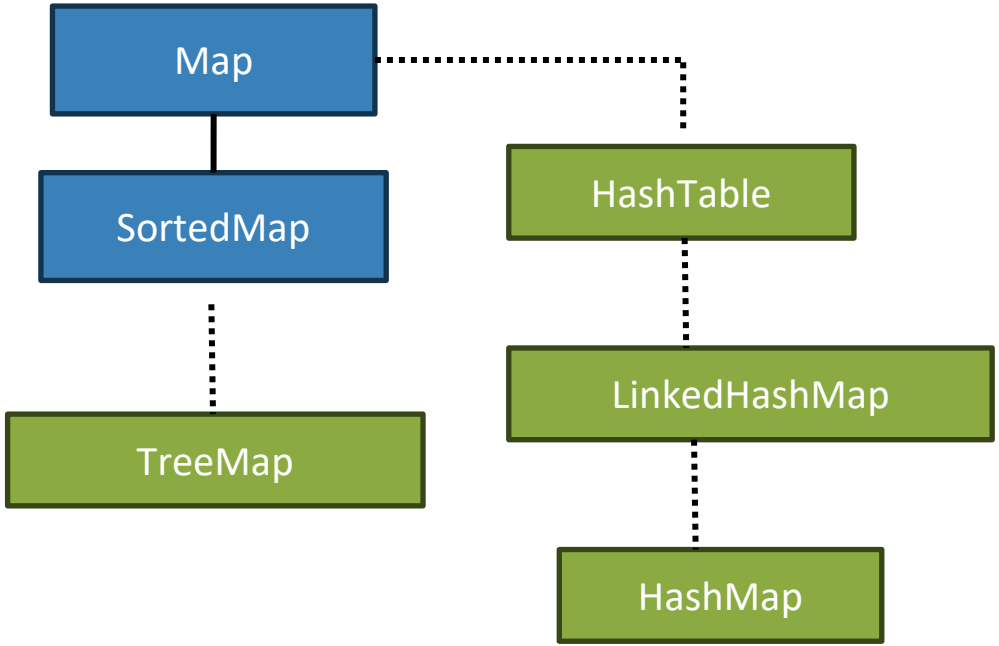
- What is **Map** interface?
- Introduction to the map interface
- HashMap key points & Example code
- LinkedHashMap key points & example code
- TreeMap key points & example code
- Comparing HashMap, LinkedHashMap, and TreeMap

WHAT IS MAP INTERFACE?

The Map interface represents a collection of key-value pairs, where each key maps to exactly one value

Keys are unique (no duplicates allowed), but values can be duplicated

INTRODUCTION TO THE MAP INTERFACE



HASHMAP KEY POINTS

- Implements the Map interface and is backed by a hash table.
- Provides constant-time performance (O(1)) for basic operations like put(), get(), and remove(), assuming a good hash function.
- Does not guarantee any order of the keys or values.
- Best suited for use cases where order is not important but fast access to data via keys is needed.
- Frequently Used Methods in **HashMap**
  - **put**(K key, V value): Inserts or updates a key-value pair.
  - **get**(Object key): Retrieves the value associated with the specified key.
  - **remove**(Object key): Removes the key-value pair for the specified key.
  - **containsKey**(Object key): Checks if the map contains the specified key.
  - **keySet**(): Returns a set of the keys contained in the map.
  - **values**(): Returns a collection of the values contained in the map.
  - **size**(): Returns the number of key-value pairs.

HASHMAP EXAMPLE

```
Account Balance of CUST001: $5000.0
Customer Accounts after removal: {CUST002=1500.0, CUST001=5000.0}
```

```java
import java.util.HashMap;
import java.util.Map;

public class BankHashMap {
   public static void main(String[] args) {
      Map<String, Double> customerAccounts = new HashMap<>();

      customerAccounts.put("CUST001", 5000.0);
      customerAccounts.put("CUST002", 1200.0);
      customerAccounts.put("CUST003", 3000.0);

      System.out.println("Account Balance of CUST001: $" + customerAccounts.get("CUST001"));

      customerAccounts.put("CUST002", 1500.0);
      customerAccounts.remove("CUST003");
      System.out.println("Customer Accounts after removal: " + customerAccounts);
   }
}
```

LINKEDHASHMAP KEY POINTS

- Implements the Map interface and maintains a linked list of entries, preserving the insertion order.
- Provides constant-time performance (O(1)) like HashMap but with the added benefit of maintaining order.
- Best suited for use cases where the order of the element's matters, such as logging user activity.
- Frequently Used Methods in **LinkedHashMap**
  - **put**(K key, V value): Inserts or updates a key-value pair while maintaining insertion order.
  - **get**(Object key): Retrieves the value associated with the specified key.
  - **remove**(Object key): Removes the key-value pair for the specified key.
  - **containsKey**(Object key): Checks if the map contains the specified key.
  - **keySet**(): Returns the keys in insertion order.
  - **values**(): Returns the values in insertion order.

LINKEDHASHMAP EXAMPLE

```
import java.util.LinkedHashMap;
import java.util.Map;

public class InsuranceLinkedHashMap {
   public static void main(String[] args) {
      Map<String, String> policyHolders = new LinkedHashMap<>();

      policyHolders.put("POL001", "John Doe");
      policyHolders.put("POL002", "Jane Smith");
      policyHolders.put("POL003", "Alice Brown");

      System.out.println("Policy Holders: " + policyHolders);

      policyHolders.put("POL002", "Jane Doe");

      for (Map.Entry<String, String> entry : policyHolders.entrySet()) {
         System.out.println("Policy: " + entry.getKey() + ", Holder: " + entry.getValue());
      }
   }
}
```

```
Policy Holders: {POL001=John Doe, POL002=Jane
Smith, POL003=Alice Brown}
Policy: POL001, Holder: John Doe
Policy: POL002, Holder: Jane Doe
Policy: POL003, Holder: Alice Brown
```

TREEMAP KEY POINTS

- Implements the NavigableMap interface (a subtype of Map), backed by a Red-Black Tree.
- Provides logarithmic time performance (O(log n)) for basic operations like put(), get(), and remove().
- Maintains the keys in sorted order (either natural order or according to a custom comparator).
- Best suited when you need a sorted map, such as maintaining a sorted record of accounts by their balances.
- Frequently Used Methods in **TreeMap**
  - **put**(K key, V value): Inserts or updates a key-value pair in sorted order.
  - **get**(Object key): Retrieves the value associated with the specified key.
  - **firstEntry**(): Returns the key-value pair with the lowest key.
  - **lastEntry**(): Returns the key-value pair with the highest key.
  - **remove**(Object key): Removes the key-value pair for the specified key.
  - **keySet**(): Returns a sorted set of the keys.
  - **values**(): Returns a collection of values in sorted order.

TREEMAP EXAMPLE

```
import java.util.TreeMap;

public class RealEstateTreeMap {
    public static void main(String[] args) {
        System.out.println("Real Estate TreeMap");
        TreeMap<Integer, String> propertyListings = new TreeMap<>();
        propertyListings.put(500000, "PROP001");
        propertyListings.put(300000, "PROP002");
        propertyListings.put(750000, "PROP003");
        System.out.println("Property Listings: " + propertyListings);
        System.out.println("Cheapest Property: " + propertyListings.firstEntry());
        System.out.println("Most Expensive Property: " + propertyListings.lastEntry());
    }
}
```

```
Property Listings: {300000=PROP002, 500000=PROP001, 750000=PROP003}
Cheapest Property: 300000=PROP002
Most Expensive Property: 750000=PROP003
```

COMPARING HASHMAP, LINKEDHASHMAP, TREEMAP

Feature	HashMap	LinkedHashMap	TreeMap
Data Structure	Hash table	Hash table + Linked List	Red-Black Tree
Search Performance	Fast (O(1))	Fast (O(1))	Moderate (O(log n))
Insertion/Deletion	Fast (O(1))	Fast (O(1))	Moderate (O(log n))
Ordering	None	Maintains insertion order	Sorted (natural or custom)
Best Use Case	Fast access with no order	Fast access with insertion order	Access sorted data

JAVA GENERICS

5

COLLECTION WITH GENERICS

Learning Objectives

- What is generics?
- Key benefits of generics
- Basic syntax of generics class & example
- Basic syntax of generics method & example
- Generics in collection & example

WHAT IS GENERICS?

Definition

- Generics in Java enable the creation of classes, interfaces, and methods that can operate on any data type while providing compile-time type safety.
- Generics allow for parameterized types, meaning the type of data a collection can store is specified at runtime.

Purpose

- Generics make code reusable and allow the use of the same class or method for different data types, avoiding the need for type casting.

KEY BENEFITS

Type Safety at Compile-time

Generics allow for stronger type checks during compilation, reducing the risk of errors at runtime.

Code Reusability

Generics allow you to write more flexible and reusable code. Instead of writing multiple versions of a class or method for different data types, you can use a generic version.

Avoiding ClassCastException

Generics eliminate the need for manual casting by ensuring type consistency, thus avoiding ClassCastException that would typically arise from type mismatches.

BASIC SYNTAX OF GENERICS CLASS

- A Generic Class allows the use of parameterized types, enabling the class to handle different data types.

- <T> is a type parameter that can be substituted with any object type (e.g., Integer, String, custom objects)

```
class Box<T> {
    private T item;
    public void setItem(T item) {
        this.item = item;
    }
    public T getItem() {
        return item;
    }
}
```

- In this example, the class Box<T> can be instantiated with different types such as Box<Integer>, Box<String>, etc.

BASIC SYNTAX OF GENERICS METHOD

- A Generic Method allows the method to accept and return any type of object.

```
public <T> T doTransaction(T transactionDetails) {
    return transactionDetails;
}
```

- Here, the doTransaction method can process transactions with various types of details (e.g., String, Object, or custom types).

EXAMPLE

- In automation testing, you may handle different types of results:
  - For a UI test, you might have a boolean result (pass or fail).
  - For an api test, the result might be a string representing the response body.

```
class TestResult<T> {
    private T result;

    public void setResult(T result) {
        this.result = result;
    }

    public T getResult() {
        return result;
    }
}
```

```
TestResult<Boolean> uiTestResult = new TestResult<>();
uiTestResult.setResult(true);  // Pass or Fail as a Boolean

TestResult<String> apiTestResult = new TestResult<>();
apiTestResult.setResult("200 OK");  // HTTP response status

TestResult<Integer> performanceTestResult = new TestResult<>();
performanceTestResult.setResult(500);  // Response in milliseconds
```

GENERICS IN COLLECTION

- Using Generics in collections ensures type safety, meaning only objects of the specified type can be added to the collection. This reduces runtime errors like ClassCastException.

```
List<String> stringList = new ArrayList<>();
stringList.add("Automation");
stringList.add("Testing");
```

- Here, List<String> ensures that only String objects can be added to the stringList. Trying to add any other data type will result in a compile-time error, preventing issues at runtime.

EXAMPLE

- Consider managing employees in a tech company, where each employee has a role such as Engineer or Manager. A type-safe collection using Generics could be

```
List<Employee> employeeList = new ArrayList<>();
  employeeList.add(new Engineer());
  employeeList.add(new Manager());
```

- This ensures that only Employee objects (or subclasses like Engineer and Manager) can be added to employeeList, maintaining consistency and preventing errors during data handling.

THANKS!

Your feedback is welcome
support@kavinschool.com