Unit 4

Java 8 features collection Framework

Java 8 is one of the most significant releases in the history of the Java programming language. Released on March 18, 2014, by Oracle. It introduced revolutionary features that transformed the way Java is used, particularly in conjunction with functional programming.

Support for functional programming with Lambda expressions and functional interfaces.

Introduction of the Stream API for efficient data processing.

A new Date and Time API for better handling of time and date operations.

Default and static methods in interfaces for greater flexibility.

4.1 Java Lambda Expressions

Lambda Expressions are anonymous functions. These functions do not need a name or a class to be used. Lambda expressions are added in Java 8. Lambda expressions express instances of functional interfaces. An interface with a single abstract method is called a functional interface. Lambda expressions implement only one abstract function and therefore implement functional interfaces.

Java lambda expressions, introduced in Java 8, allow developers to write concise, functional-style code by representing anonymous functions. They enable passing code as parameters or assigning it to variables, resulting in cleaner and more readable programs.

Lambda expressions implement a functional interface (An interface with only one abstract function)

Enable passing code as data (method arguments).

Allow defining behaviour without creating separate classes.

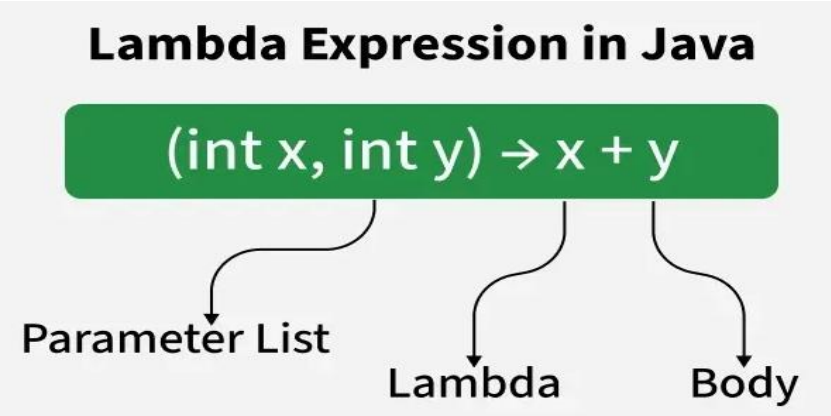
```
interface Add
{
      int addition(int a, int b);
}
public class java8{
   public static void main(String[] args){
            // Lambda expression to add two numbers
      Add add = (a, b) -> a + b;
            int result = add.addition(10, 20);
      System.out.println("Sum: " + result);
 }
}
```

Output

Sum: 30

4.2 Syntax of Lambda Expressions

Java Lambda Expression has the following syntax:



Parameter List: Parameters for the lambda expression

Arrow Token (->): Separates the parameter list and the body

Body: Logic to be executed.

4.3 Functional interface

A functional interface has exactly one **abstract method**. Lambda expressions provide its implementation. @FunctionalInterface annotation is optional but recommended to enforce this rule at compile time.

```
interface FuncInterface{
      void abstractFun(int x);
   default void normalFun(){
      System.out.println("Hello");
      }
}
public class Java8{
    public static void main(String[] args){
            FuncInterface fobj = (int x) -> System.out.println(2 * x);
             fobj.abstractFun(5);
   }
```

```
}
```

Output

10

4.4 Types of Lambda Parameters

There are three Lambda Expression Parameters are mentioned below:

1. Lambda with Zero Parameters

Syntax:

```
() -> System.out.println("Zero parameter lambda");
```

```
@FunctionalInterface
interface ZeroParameter{
    void display();
}


public class Java8{
    public static void main(String[] args){
        // Lambda expression with zero parameters
ZeroParameter zeroParamLambda = ()-> System.out.println("This is a zero-parameter lambda expression!");
    // Invoke the method
    zeroParamLambda.display();
  }
}
```

Output

This is a zero-parameter lambda expression!

2. Lambda with a Single Parameter

Syntax:

```
(p) -> System.out.println("One parameter: " + p);
```

It is not mandatory to use parentheses if the type of that variable can be inferred from the context.

```
@FunctionalInterface
interface Greeting {
  void sayHello(String name);
```

```
}


public class Main {
   public static void main(String[] args) {
      Greeting greet = name -> System.out.println("Hello, " + name + "!");
      greet.sayHello("MIT WPU");
   }
}
```

Output:

```
Hello, MIT WPU!
```

⚙ Rules to Remember

Parentheses optional if there's one parameter and no type declared:

✔ *x -> x + 1*

✘ *(x, y) -> x + y — multiple parameters need parentheses.*

Braces {} required if you have multiple statements:

```
x -> {
   int y = x + 2;
   return y * y;
};
```

The parameter type can be inferred, but you can include it explicitly if you wish:

3. Lambda Expression with Multiple Parameters

Syntax:

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
@FunctionalInterface
interface Functional {
   int operation(int a, int b);
}
public class Java8 {
   public static void main(String[] args) {
      // Using lambda expressions to define the operations
      Functional add = (a, b) -> a + b;
```

```
        Functional multiply = (a, b) -> a * b;
        // Using the operations
        System.out.println(add.operation(6, 3));
        System.out.println(multiply.operation(4, 5));
    }
}
```

Output

9

20

4.5 Java Functional Interfaces

A functional interface in Java is an interface that **contains only one abstract method**.

Functional interfaces can have multiple default or static methods, but only one abstract method.

From Java 8 onwards, **lambda expressions and method references** can be used to represent the instance of a functional interface.

Example: Using a Functional Interface with Lambda Expression

```
public class MIT {
    public static void main(String[] args) {
        // Using lambda expression to implement Runnable
        new Thread(() -> System.out.println("New thread created")).start();
    }
}
```

Output

New thread created

Explanation:

Above program demonstrates use of lambda expression with the Runnable functional interface.

Runnable has one abstract method run(), so it qualifies as a functional interface.

Lambda ()-> System.out.println("New thread created") defines the run() method.

new Thread().start() starts a new thread that executes the lambda body

Note:

*A **f**unctional interface can also extend another functional interface.*

@FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface cannot have more than one abstract method. In case more than one abstract methods are present, the compiler flags an "Unexpected @FunctionalInterface annotation" message. However, it is not mandatory to use this annotation.

Note:

@FunctionalInterface annotation is optional but it is a good practice to use. It helps catching the error in early stage by making sure that the interface has only one abstract method.

Example: Defining a Functional Interface with @FunctionalInterface Annotation

```
@FunctionalInterface
interface Square {
   int calculate(int x);
}
class MIT {
   public static void main(String args[]) {
      int a = 5;
      // lambda expression to define the calculate method
      Square s = (int x) -> x * x;
      // parameter passed and return type must be same as defined in the prototype
      int ans = s.calculate(a);
      System.out.println(ans);
   }
}
```

Output

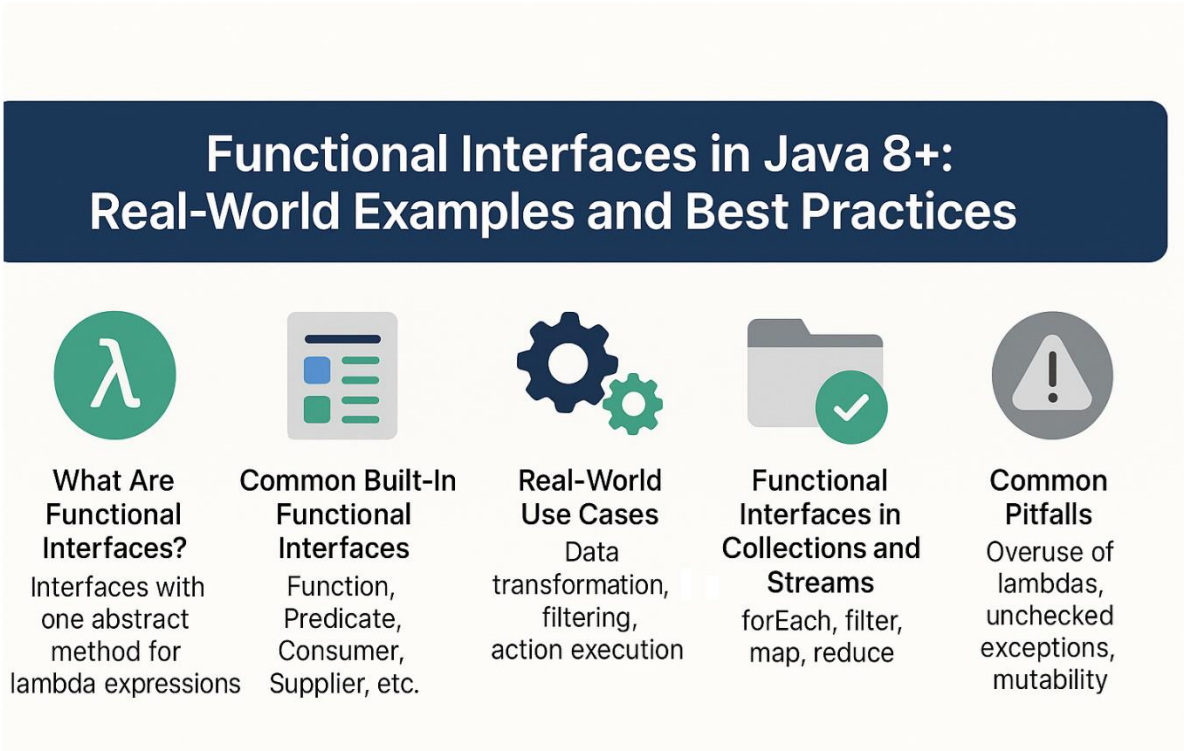
25

Explanation:

Square is a functional interface with a single method calculate(int x).

A lambda expression (int x) -> x * x is used to implement the calculate method.

Lambda takes x as input and returns x * x.



Java 8 has defined a lot of functional interfaces to be used extensively in lambda expressions. Following is the list of functional interfaces defined in java.util.Function package.

4.6 Types of Functional Interfaces in Java

There are primarily four types of functional interfaces in Java.

Predicate

Consumer

Supplier

Function

4.6.1 Predicate Functional Interface

A predicate functional interface is the one whose method accepts one argument and will return either true or false. A predicate functional interface is mainly used in comparison to sort elements or to filter a value based on certain condition(s) applied on the input passed.

Syntax

```
Predicate predicate = (value) -> value  != 0;

// or

Predicate predicate = (value) -> test(value);
```

In above snippet predicate function is to return either true/false based on the value passed.

Example

In this example, were using a predicate functional interface to filter odd numbers from a list of integers with the help of a lambda expression.

```
import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

public class Tester {

  public static void main(String args[]) {

    List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8);

    Predicate<Integer> isEvenNumber = n -> n %2 == 0;

    numbers =  numbers.stream().filter(isEvenNumber).toList();

    System.out.println(numbers);

  }

}
```

This will produce the following result –

```
[2, 4, 6, 8]
```

4.6.2 Consumer Functional Interface

A consumer functional interface is one whose method accepts one argument and will not return anything. A consumer functional interface is mainly used for side-effect operations. For example, to print an element, to add a salutation, etc..

Syntax

```
Consumer consumer = (value) -> System.out.println(value);

// Or

Consumer consumer1 = System.out::println;

// Or

Consumer consumer2 = (value) -> accept(value);
```

Example

In this example, were using consumer functional interface to print all numbers from a list of integers with the help of a lambda expression and a method reference.

```
import java.util.function.Consumer;

Consumer<String> printMessage = msg -> System.out.println(msg);
```

printMessage.accept("Welcome to Java 8!");

4.6.3 Supplier Functional Interface

The supplier functional interface is the one whose method does not have any arguments to pass and will return a value. A supplier functional interface is mainly used to generate values lazily. For example, to get a random number, to generate a sequence of numbers, etc.

Syntax

```
Supplier supplier = () -> Math.random() * 10;

// or

Supplier supplier1 = () -> get();
```

Example

```
import java.util.function.Supplier;

Supplier<String> getGreeting = () -> "Hello, World!";

String greeting = getGreeting.get();

System.out.println(greeting);
```

4.6.4 Function Functional Interface

Function functional interface is one whose method **accepts one argument and will return a value.**

A function functional interface is mainly used to get the processed value. For example, to get the square of an element, to trim string values, etc.

Syntax

```
Function function = (value) -> Math.random() * 10;

// or

Function function1 = (value) -> apply(value);
```

Example

```
import java.util.function.Function;

Function<String, Integer> stringToLength = str -> str.length();

int length = stringToLength.apply("Hello");

System.out.println("The length of 'Hello' is: " + length);
```

Functional Interfaces Review

Functional Interfaces	Description	Method
Consumer	It accepts a single input argument and returns no result.	void accept(T t)
Predicate	It accepts a single argument and returns a boolean result.	boolean test(T t)
Function	It accepts a single argument and returns a result.	R apply(T t)
Supplier	It does not take any arguments but provides a result.	T get()

4.7 JODA-Time

Joda-Time is an API created by joda.org which offers better classes and having efficient methods to handle date and time than classes from java.util package like Calendar, Gregorian Calendar, Date, etc. This API is included in Java 8.0 with the java.time package. To include we need to import following:

import java.time.*;

4.7.1 Basic Features of JODA-TIME

* It uses easy field accessors like getYear(), getDayOfWeek(), getDayofYear().

* It supports 7 Calendar Systems like Buddhist, Coptic, Ethiopic, Gregorian, GregorianJulian, Islamic, Julian.

* There is a Provision to create our own Calendar system.

* It Provides rich Set of Methods for date and Time calculations.

* It uses a database for time zones. This database updated manually several times in a year.

* Its Methods executes faster compared to earlier methods of java 7.0; thus, provides better performance

* Its Objects are immutable. So, they are thread-safe

4.7.2 Important Classes in java.time package.

1. **DateTime** : Immutable replacement for JDK Calendar.

DateTime dt = new DateTime(); // which creates a datetime object representing the current date and time in milliseconds as determined by the system clock. It is constructed using the ISO calendar in the default time zone.

2. **LocalDate** : This class represents a date in the form of year-month-day and useful for representing a date without time and time

zone.LocalDate today = LocalDate.now() //gives System date into LocalDate object using now method.

System.out.println(today) // 2025-11-08

int d = today.getDayOfMonth(); // 11

3. **LocalTim**e : This class represents the time of the day without time

zone.LocalTime time = LocalTime.now(); //gives System time into localTime object

System.out.println(time); // 10:19:58

4. **LocalDateTime** : This class handles both date and time without considering the time zone. get current date and time

LocalDateTime dt = LocalTime.now();

4.7.3 Setting up Environment

1. Create your java project in eclipse.

2. Download the latest JodaTime .tar.gz file Click Here, and extract its contents.

3. In Eclipse, look for your project at package explorer and right click on it then call it New -> Folder -> libs

4. Copy/Drag joda-time-2.1.jar into the new created libs folder.

5. Right click on your project again (in package explorer) then Properties -> Java Build Path -> Libraries -> Add Jars -> joda-time-2.1.jar

6. Now you can test with this code : DateTime test = new DateTime();

Basic Example

```
// Java program to illustrate

// functions of JODA time

import org.joda.time.DateTime;

import org.joda.time.LocalDateTime;

public class JodaTime {

    public static void main(String[] args)

    {
```

```
        DateTime now = new DateTime();
        System.out.println("Current Day: " + now.dayOfWeek().getAsText());
        System.out.println("Current Month: " + now.monthOfYear().getAsText());
        System.out.println("Current Year: " + now.year().getAsText());
        System.out.println("Current Year is Leap Year: " + now.year().isLeap());
        // get current date and time
        LocalDateTime dt = LocalDateTime.now();
        System.out.println(dt);
    }
}
```

4.7.4 Advantages

* Similar usage across multiple Java platforms.

* Supports additional calendars such as Buddhist and Ethiopic.

* Self-reported better performance.

* Easy interoperability: The library internally uses a millisecond instant which is identical to the JDK and similar to other common time representations. This makes interoperability easy, and Joda-Time comes with out-of-the-box JDK interoperability.

4.7.5 Disadvantages:

* Requires installation of package and perhaps updates from Joda.org.

Note: JODA is now not supported by latest version of jdk.

So, for practicals you can show the execution of a program without JODA

4.8 The Java Collection Framework

It is a set of interfaces and classes in the package that provides a unified architecture for storing and manipulating groups of objects. It includes pre-built data structures like lists, sets, and maps, and offers methods for common operations like sorting, searching, and synchronization. This framework improves developer productivity by offering ready-to-use, dynamic, and high-performance implementations instead of requiring developers to build them from scratch.

What is the Java Collection Framework?

A set of interfaces and classes in the `java.util` package.

Provides a unified architecture for handling groups of objects.

Includes pre-built data structures like lists, sets, and maps.

Offers common operations such as sorting, searching, and synchronization.

Enhances developer productivity through ready-to-use, efficient components.

4.8.1 Key Features and Benefits

Standardized architecture:
Consistent design for handling different collection types.

Ready-made data structures:
High-performance implementations like List, Set, and Map.

Dynamic size:
Collections can grow or shrink as needed (unlike arrays).

Built-in algorithms:
Includes utilities for sorting, searching, reversing, etc.

Improved interoperability:
Common language for passing collections between APIs.

Enhanced performance:

Easily switch between implementations to tune performance •

4.8.2 Core components

* Interfaces: These define the contracts for collection types

* Implementations: These are the concrete classes that provide the actual data structures based on the interfaces

* Algorithms: These are the methods that perform operations on the collections, such as sorting or searching

4.8.3 Advantages:

The Java Collection Framework provides a **standardized, dynamic, and efficient** way to manage groups of objects.

It simplifies coding, improves performance, and promotes reusability.

Essential for building scalable and maintainable Java applications.

4.9 List Interface

Belongs to the **java.util** package

Represents an **ordered collection** of elements

Extends the **Collection** interface

Provides additional functionality for managing **ordered sequences**

4.9.1 Key Characteristics of List Interface

Ordered Collection: Maintains insertion order

Indexed Access: Elements accessed via integer index (starting from 0)

Allows Duplicates: Multiple identical elements permitted

Allows Null Elements: Can store null values

4.9.2 Commonly Used Methods

add(E e) – Appends element to end

add(int index, E element) – Inserts at specified position

get(int index) – Returns element at index

set(int index, E element) – Replaces element at index

remove(int index) – Removes element at index

remove(Object o) – Removes first occurrence of element

indexOf(Object o) – Returns first occurrence index

lastIndexOf(Object o) – Returns last occurrence index

size() – Returns number of elements

isEmpty() – Checks if list is empty

clear() – Removes all elements

4.9.3 Common Implementation of List Interface

ArrayList: Resizable array, fast random access

LinkedList: Doubly-linked, efficient for insertion/deletion

Vector: Legacy, synchronized version of ArrayList

Stack: LIFO implementation extending Vector

4.10 Introduction to ArrayList

A **resizable array implementation** of List

Located in **java.util** package

Overcomes fixed-size limitation of arrays

4.10.1 Key Characteristics of ArrayList

- **Resizable:** Automatically grows/shrinks
- **Indexed Access:** Access elements via index

- **Allows Duplicates** and **Maintains Order**
- **Not Synchronized** (requires manual sync for multithreading)
- **Stores Objects:** Uses wrapper classes for primitives

4.10.2 Creating and Using ArrayList

Import: Begin by importing the ArrayList class:

```
import java.util.ArrayList;
```

Declaration and Instantiation:

Declare an ArrayList and specify the type of elements it will hold using generics:

```
ArrayList<String> names = new ArrayList<String>(); // For Strings
ArrayList<Integer> numbers = new ArrayList<Integer>(); // For Integers
```

Adding Elements:

Use the add() method to add elements:

```
names.add(“MIT");    names.add(“WPU");
```

Accessing Elements:

Use the get() method with an index to retrieve elements:

```
String firstPerson = names.get(0); // Retrieves “MIT“
```

Removing Elements: Use the remove() method, either by index or by specifying the object itself:

```
names.remove(0); // Removes “MIT"    names.remove(“WPU"); // Removes “WPU“
```

Iterating: Elements can be iterated using *a for-each loop or an Iterator.*

4.10.3 Programming Example

```
import java.util.ArrayList;
public class ArrayListExample {
  public static void main(String[] args) {
ArrayList<String> f = new ArrayList<String>();
f.add(“MIT");
 f.add(“WPU");
```

```
   f.add(“PUNE");
System.out.println(“Elements in the list: " + fruits);
  String arr = f.get(0);
 System.out.println(“Elements: " + arr);
   f.remove(“PUNE");
   System.out.println(“Elements after removing PUNE: " + f);    }}
```

4.11 LinkedList

LinkedList is a part of the Java Collection Framework and is present in the java.util package. It implements a doubly-linked list data structure where elements are not stored in contiguous memory. Each node contains three parts: the data, a reference to the next node, and a reference to the previous node

- **Dynamic Size:** LinkedList grows or shrinks dynamically at runtime.
- **Maintains Insertion Order:** Elements are stored in the order they are added.
- **Allows Duplicates:** Duplicate elements are allowed.
- **Not Synchronized:** By default, LinkedList is not thread-safe. To make Thread-safe use of Collections.synchronizedList().
- **Efficient Insertion/Deletion:** Adding or removing elements at the beginning or middle is faster compared to ArrayList.

4.11.1 Programming Example

```
import java.util.LinkedList;
public class Mit {
   public static void main(String[] args){
         // Creating a LinkedList
     LinkedList<String> l = new LinkedList<String>();
     // Adding elements to the LinkedList using add() method
     l.add("One");
     l.add("Two");
     l.add("Three");
     l.add("Four");
     l.add("Five");
```

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

Output

[One, Two, Three, Four, Five]

Explanation:

- Creates an empty LinkedList of Strings.
- Adds elements "One" to "Five" using the add() method.
- Prints the LinkedList showing elements in insertion order: [One, Two, Three, Four, Five].

Note: *LinkedList nodes cannot be accessed directly by index; elements must be accessed by traversing from the head.*

4.11.2 Programming Example 2

```
import java.util.*;
public class Mit {
 public static void main(String args[])
  {
  LinkedList<String> ll = new LinkedList<>();
  ll.add("Mit");
 ll.add("Mit");
 ll.add(1, "For");
 System.out.println("Initial LinkedList " + ll);
 // Function call
ll.remove(1);
 System.out.println("After the Index Removal " + ll);
 ll.remove("Mit");
System.out.println("After the Object Removal "  + ll);
}}
```

Output

Initial LinkedList [MIt, For, Mit

After the Index Removal [Mit, Mit]

After the Object Removal [Mit]

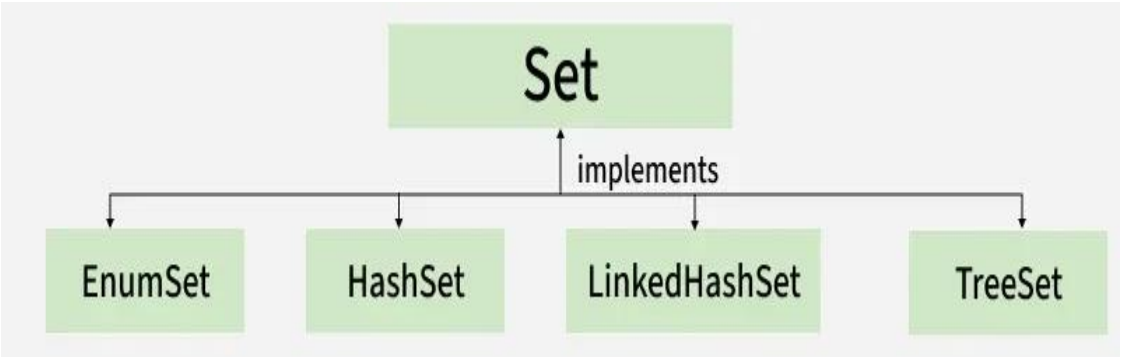
4.12 Set Interface

In Java, the Set interface is a part of the Java Collection Framework,

located in the java.util package.

It represents a collection of unique elements, meaning it does not allow duplicate values.

- The set interface does not allow duplicate elements.
- It can contain at most one null value except TreeSet implementation which does not allow null.
- The set interface provides efficient search, insertion, and deletion operations.
- **Hierarchy of Java Set interface**
  The image below demonstrates the hierarchy of Java Set interface.



Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access.

```
import java.util.HashSet;
import java.util.Set;
public class Demo {
    public static void main(String args[])
  {
    // Create a Set using HashSet
    Set<String> s = new HashSet<>();
    // Displaying the Set
    System.out.println("Set Elements: " + s);
  }
}
```

Explanation: In the above example, HashSet will appear as an empty set, as no elements were added. The order of elements in HashSet is not guaranteed, so the elements will be displayed in a random order if any are added.

Classes that implement the Set interface

- **HashSet:** A set that stores unique elements without any specific order, using a hash table and allows one null element.
- **EnumSet** : A high-performance set designed specifically for enum types, where all elements must belong to the same enum.
- **LinkedHashSet**: A set that maintains the order of insertion while storing unique elements.
- **TreeSet**: A set that stores unique elements in sorted order, either by natural ordering or a specified comparator.

Declaration of Set Interface

The declaration of Set interface is listed below:

public interface Set extends Collection

Creating Set Objects

Since Set is an interface, objects cannot be created of the typeset. We always need a class that extends this list in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Set. This type-safe set can be defined as:

// Obj is the type of the object to be stored in Set
Set<Obj> set = new HashSet<Obj> ();

Performing Various Operations on Set

Set interface provides commonly used operations to manage unique elements in a collection. Now let us discuss these operations individually as follows:

1. Adding Elements

To add elements to a Set in Java, use the add() method.

```
import java.util.*;


// Main class
class Demo {
    public static void main(String[] args)
   {
      Set<String> s = new HashSet<String>();
      s.add("B");
      s.add("B");
```

```
        s.add("C");
        s.add("A");
        System.out.println(s);
    }}
```

Output

[A, B, C]

2. Accessing the Elements

After adding the elements, if we wish to access the elements, we can use inbuilt methods like contains().

```
import java.util.*;

class Demo {

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

        h.add("A");
        h.add("B");
        h.add("C");
        h.add("A");

        System.out.println("Set is " + h);
```

```
        String s = "D";

        System.out.println("Contains " + s + " " + h.contains(s));
    }
}
```

Output

```
Set is [A, B, C]
Contains D false
```

3. Removing Elements

The values can be removed from the Set using the remove() method.

```
import java.util.*;
class Demo {

    public static void main(String[] args)
    {
        // Declaring object of Set of type String
        Set<String> h = new HashSet<String>();

        // Elements are added using add() method, Custom input
elements
        h.add("A");
        h.add("B");
```

```
        h.add("C");
        h.add("B");
        h.add("D");
        h.add("E");

        System.out.println("Initial HashSet " + h);

        // Removing custom element using remove() method
        h.remove("B");
        System.out.println("After removing element " + h);
    }
}
```

Output

Initial HashSet [A, B, C, D, E]

After removing element [A, C, D, E]

4. Iterating elements

There are various ways to iterate through the Set. The most famous one is to use the enhanced for loop.

```
import java.util.*;
class Demo {
    public static void main(String[] args)
    {
        // Creating object of Set and declaring String type
```

```
        Set<String> h = new HashSet<String>();
        // Adding elements to Set using add() method, Custom input
elements
        h.add("A");
        h.add("B");
        h.add("C");
        h.add("B");
        h.add("D");
        h.add("E");

        // Iterating through the Set via for-each loop
        for (String value : h)

            // Printing all the values inside the object
            System.out.print(value + ", ");

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

Output

A, B, C, D, E,

4.12.1 Methods of Set Interface

Let us discuss methods present in the Set interface provided below in a tabular format below as follows:

Method	Description
add(element)	Adds element if not already present. Returns true if added.
addAll(collection)	Adds all elements from the given collection.
clear()	Removes all elements from the set.
contains(element)	Checks if the set contains the specified element.
containsAll(collection)	Checks if the set contains all elements from the given collection.
hashCode()	Returns the hash code of the set.
isEmpty()	This method is used to check whether the set is empty or not.
iterator()	This method is used to return the iterator of the set.
remove(element)	Removes the specified element from the set.

Method	Description
removeAll(collection)	Removes all elements in the given collection from the set.
retainAll(collection)	Retains only elements present in the given collection.
size()	Returns the number of elements in the set.
toArray()	This method is used to form an array of the same elements as that of the Set.