EXCEPTION HANDLING

Presented By : Prof.Honey Parmar

EXCEPTION HANDLING IN JAVA

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

What is Exception in Java?

an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

EXCEPTION HANDLING IN JAVA

Parul® University

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

EXCEPTION HANDLING IN JAVA

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

EXCEPTION HANDLING IN JAVA

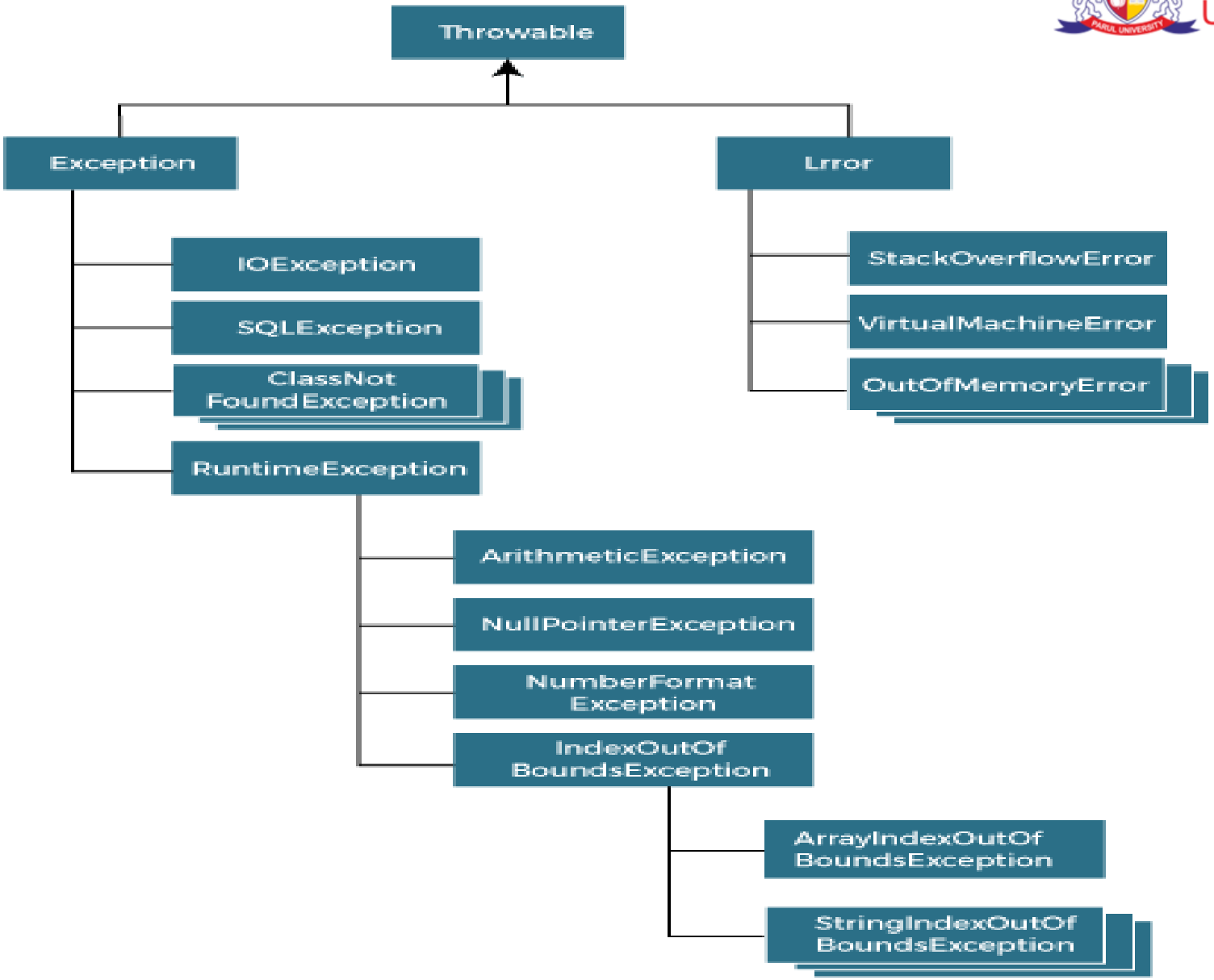
Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed.

However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

EXCEPTION HANDLING IN JAVA

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:


Throwable
Exception
Lrror
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

EXCEPTION HANDLING IN JAVA

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception.

Checked Exception
Unchecked Exception

EXCEPTION HANDLING IN JAVA

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

EXCEPTION HANDLING IN JAVA

Java Exception Keywords

1. try

The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

2. catch

The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

EXCEPTION HANDLING IN JAVA

3. finally

The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

4. throw

The "throw" keyword is used to throw an exception.

5. throws

The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

EXCEPTION HANDLING IN JAVA

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

EXCEPTION HANDLING IN JAVA

2) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

EXCEPTION HANDLING IN JAVA

Java try-catch block
Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

EXCEPTION HANDLING IN JAVA

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

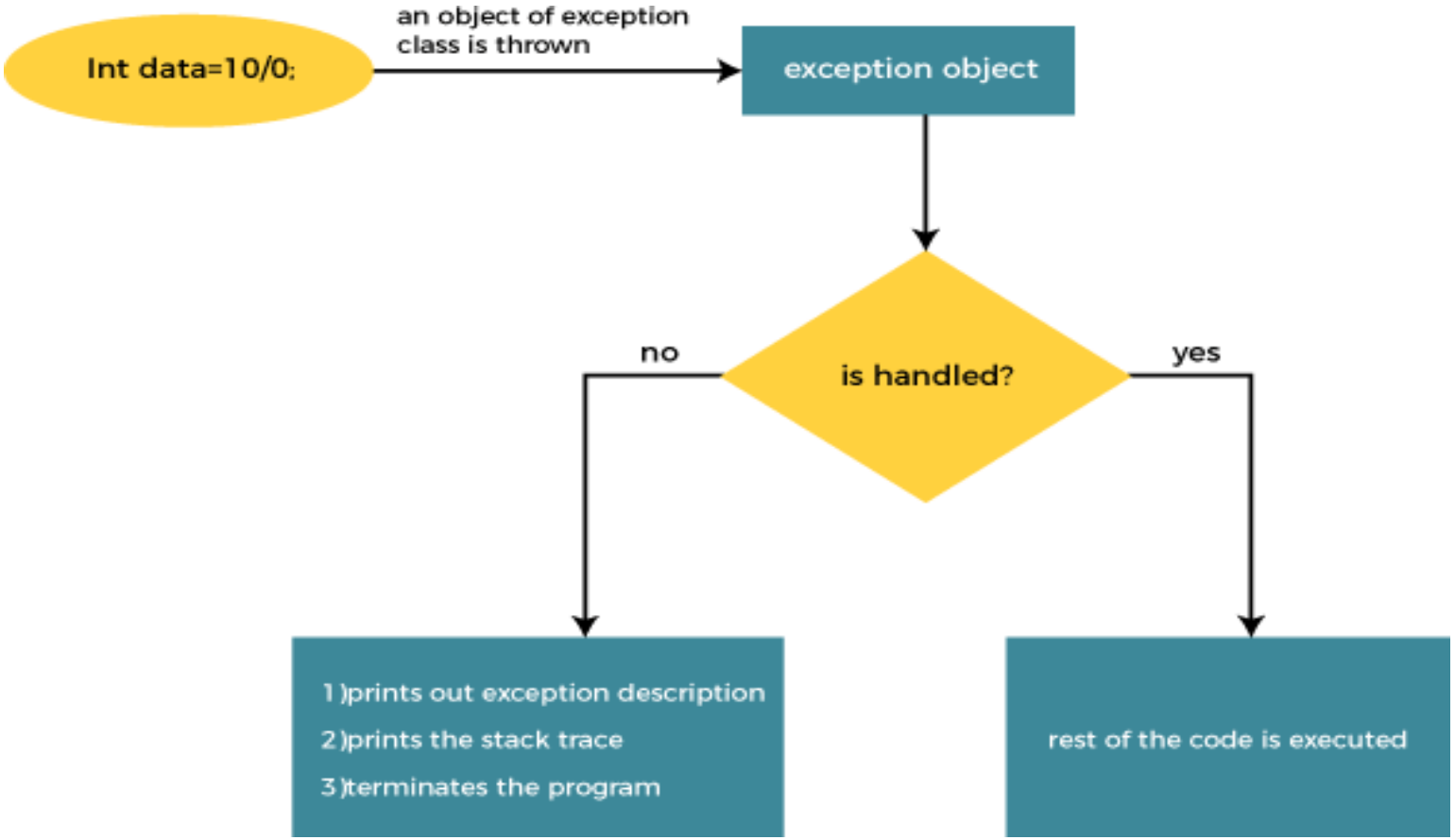
EXCEPTION HANDLING IN JAVA

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
The catch block must be used after the try block only. You can use multiple catch block with a single try block.

EXCEPTION HANDLING IN JAVA

Internal Working of Java try-catch block



EXCEPTION HANDLING IN JAVA

Problem without exception handling

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

EXCEPTION HANDLING IN JAVA

Solution by exception handling

```
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

```
java.lang.ArithmeticException: / by zero
rest of the code
```

EXCEPTION HANDLING IN JAVA

Parul® University

print a custom message on exception

```
public class TryCatchExample5 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
             // handling the exception
        catch(Exception e)
        {
                  // displaying the custom message
            System.out.println("Can't divided by zero");
        }
    }
}
```

```
Can't divided by zero
```

EXCEPTION HANDLING IN JAVA

resolve the exception in a catch block

```
public class TryCatchExample6 {

    public static void main(String[] args) {
        int i=50;
        int j=0;
        int data;
        try
        {
        data=i/j; //may throw exception
        }
            // handling the exception
        catch(Exception e)
        {
             // resolving the exception in catch block
            System.out.println(i/(j+2));
        }
    }
}
```

```
25
```

EXCEPTION HANDLING IN JAVA

handle another unchecked exception

```
public class TryCatchExample9 {

    public static void main(String[] args) {
        try
        {
        int arr[]= {1,3,5,7};
        System.out.println(arr[10]); //may throw exception
        }
            // handling the array exception
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }
}
```

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

EXCEPTION HANDLING IN JAVA

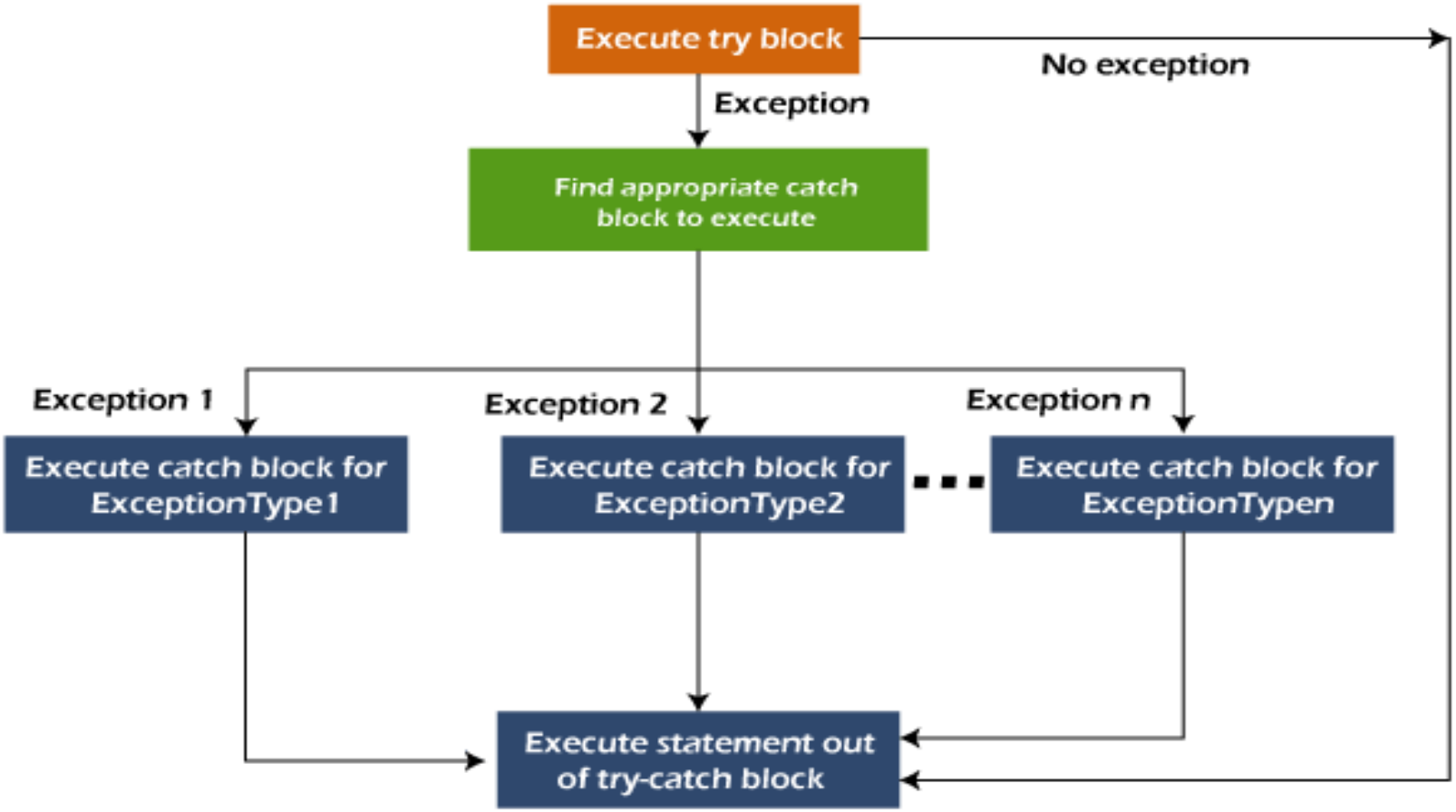
Java Catch Multiple Exceptions

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

EXCEPTION HANDLING IN JAVA

Flowchart of Multi-catch Block



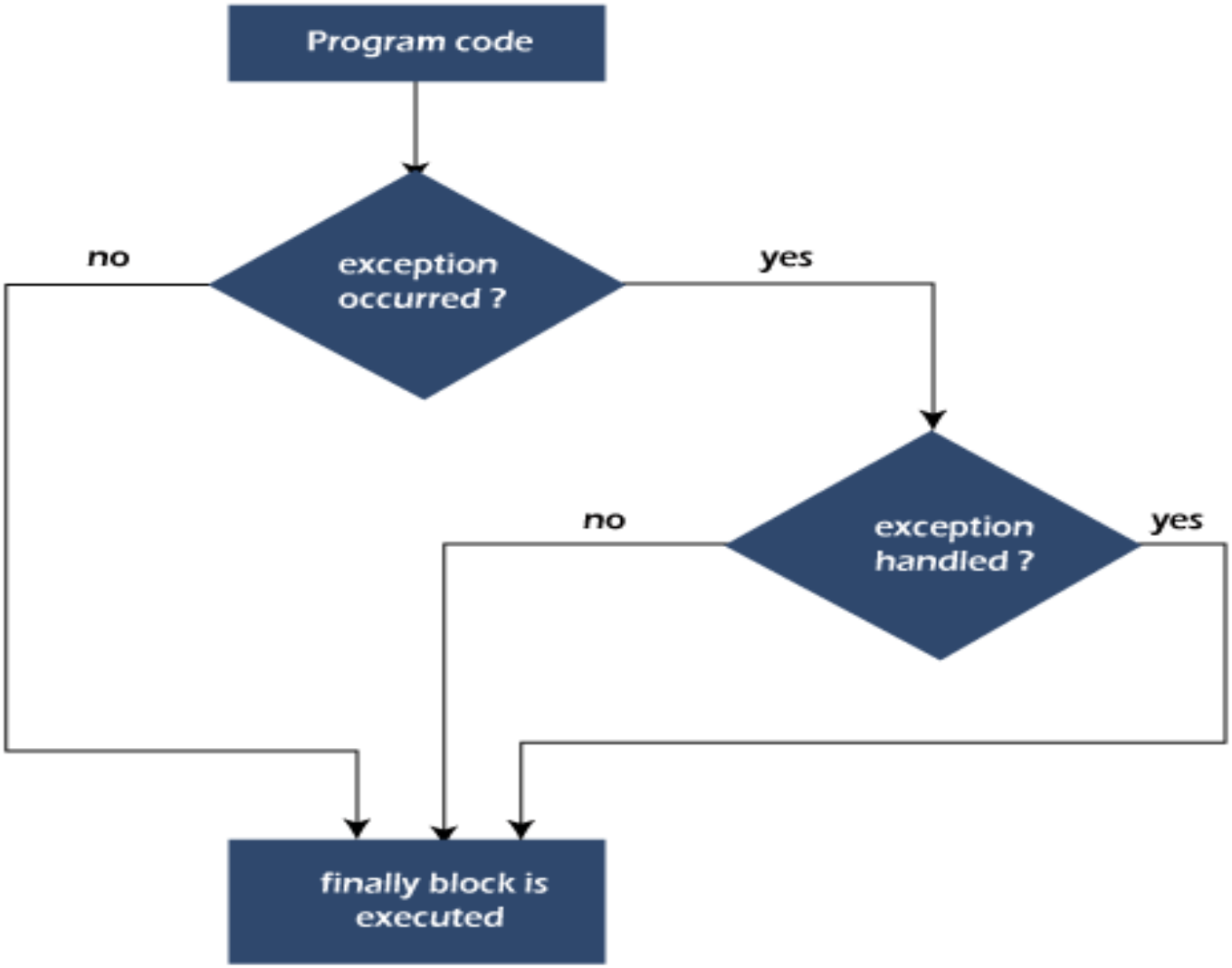
EXCEPTION HANDLING IN JAVA

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

EXCEPTION HANDLING IN JAVA



CASE 1: WHEN AN EXCEPTION DOES NOT OCCUR

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
    int data=25/5;
    System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of the code...");
  }
}
```

```
5
finally block is always executed
rest of the code...
```

CASE 2: WHEN AN EXCEPTION OCCURR BUT NOT HANDLED BY THE CATCH BLOCK

```
public class TestFinallyBlock1{
      public static void main(String args[]){

      try {
              System.out.println("Inside the try block");
             //below code throws divide by zero exception
              int data=25/0;
             System.out.println(data);
        }
      //cannot handle Arithmetic type exception
      //can only accept Null Pointer type exception
      catch(NullPointerException e){
        System.out.println(e);
      }
     //executes regardless of exception occured or not
      finally {
        System.out.println("finally block is always executed");
      }

      System.out.println("rest of the code...");
      }
}
```

```
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

EXCEPTION HANDLING IN JAVA

Parul® University

throw Exception

throw keyword is used to throw an exception explicitly.

We specify the exception object which is to be thrown. The Exception has some message with it that provides the error description.

We can throw either checked or unchecked exceptions in Java by throw keyword.

EXCEPTION HANDLING IN JAVA

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

```
throw new exception_class("error message");
```

Example

```
throw new IOException("device error");
```

EXCEPTION HANDLING IN JAVA

Throwing Unchecked Exception

we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

EXCEPTION HANDLING IN JAVA

```
public class TestThrow1 {
    //function to check if person is eligible to vote or not
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
        }
        else {
            System.out.println("Person is eligible to vote!!");
        }
    }
    //main method
    public static void main(String args[]){
        //calling the function
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to vote
        at TestThrow1.validate(TestThrow1.java:6)
        at TestThrow1.main(TestThrow1.java:15)
```

EXCEPTION HANDLING IN JAVA

Throwing checked Exception

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

EXCEPTION HANDLING IN JAVA

```
import java.io.*;
public class TestThrow2 {

    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {

        FileReader file = new FileReader("C:\\Users\\Vivek\\Desktop\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);
        throw new FileNotFoundException();
    }
    //main method
    public static void main(String args[]){
        try
        {
            method();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
    }
}
```

EXCEPTION HANDLING IN JAVA

throws keyword

Java throws keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
    //method code
    }
```

EXCEPTION HANDLING IN JAVA

```
import java.io.*;
class M{
 void method()throws IOException{
   throw new IOException("device error");
  }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();
  }
}
```

Parul® University

Collection Classes

Presented By: Prof.Honey Parmar

Parul® University

Java Collections

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList PriorityQueue, LinkedHashSet, TreeSet).

Java Collections

- **What is a framework in Java**
- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

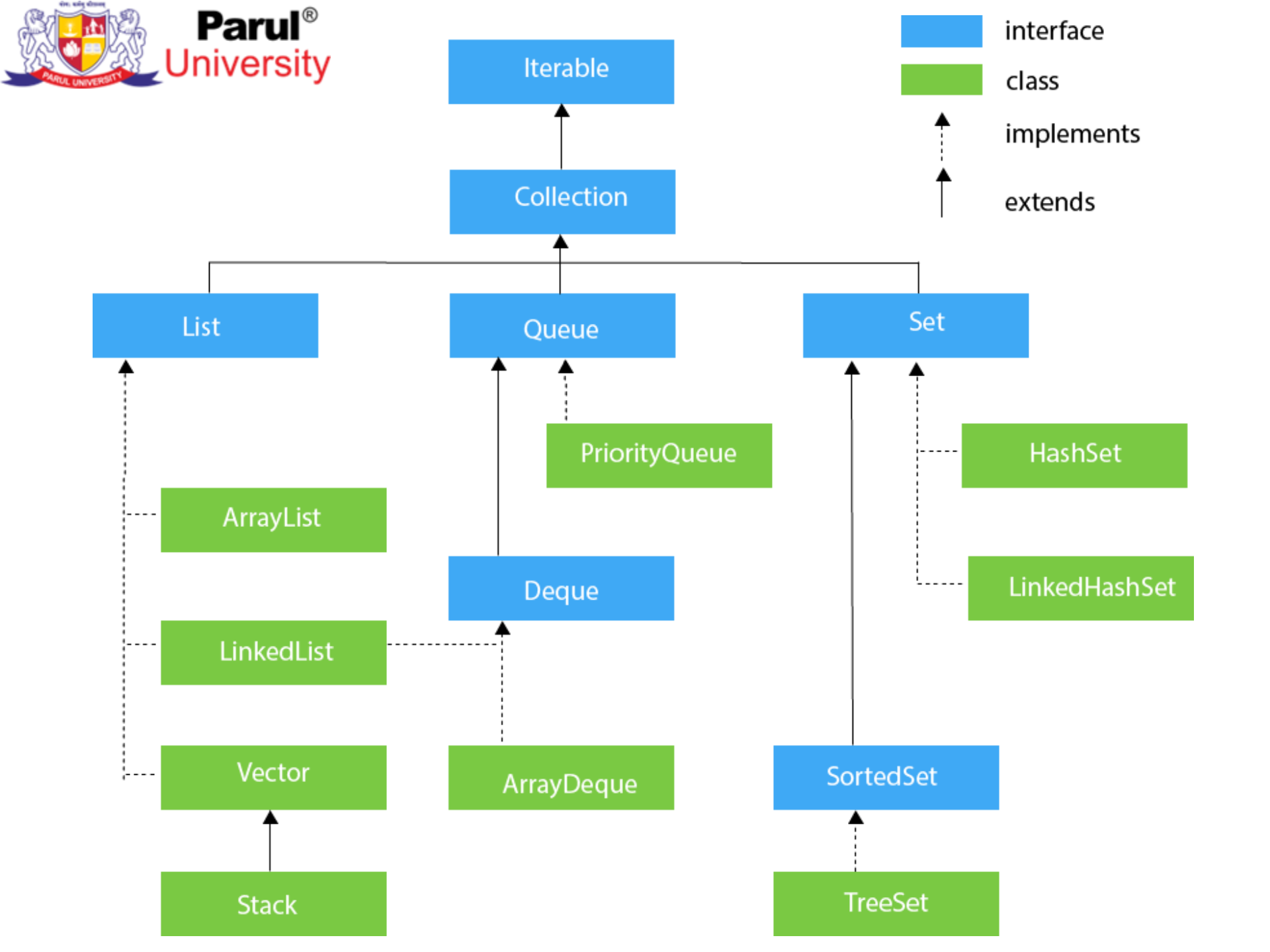
Java Collections

- **What is Collection framework**
- The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

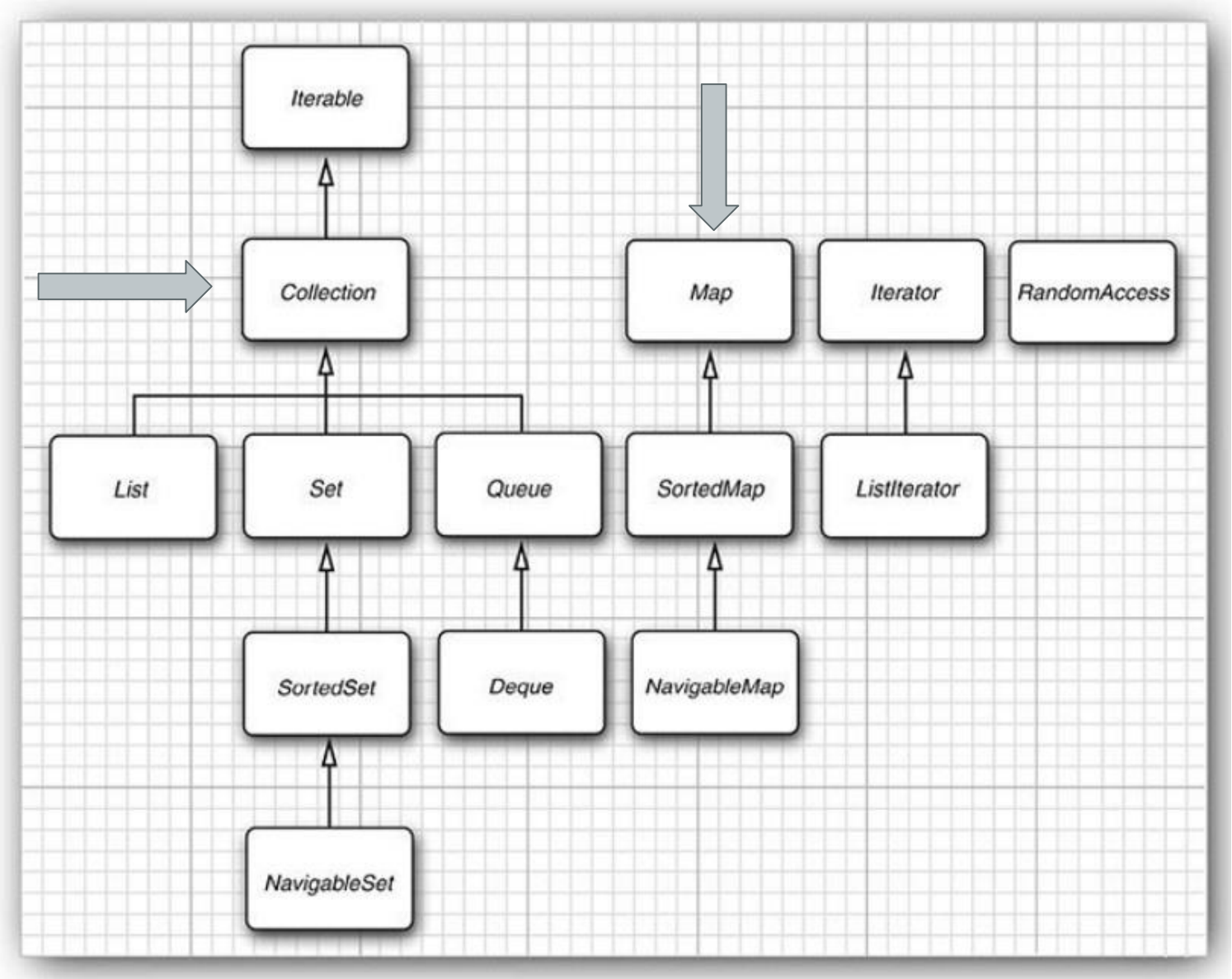
1. Interfaces and its implementations, i.e., classes
2. Algorithm

Java Collections

- **Hierarchy of Collection Framework**
- Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.


Parul®
University
Iterable
Collection
List
Queue
Set
interface
class
implements
extends
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Interfaces in Collecti Framework



Java Collections

- The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
- **Methods of Collection interface**
- There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.

7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.

15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

- Iterator interface provides the facility of iterating the elements in a forward direction only.

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

- The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
- It contains only one abstract method. i.e.,
- Iterator<T> iterator()
- It returns the iterator over the elements of type T.

Java List

- **List** in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface.
- The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

Java List

- **List** in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface.
- It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions.
- The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

Java List Methods

void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.

int hashcode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.

boolean contains(Object o)	It returns true if the list contains the specified element
boolean containsAll(Collection<?> c)	It returns true if the list contains all the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.

Java List Methods

void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

```
//Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();

//Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();
```

Java List Example

```
import java.util.*;
public class ListExample1{
public static void main(String args[]){
 //Creating a List
 List<String> list=new ArrayList<String>();
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);

}
}
```

```
Mango
Apple
Banana
Grapes
Press any key to continue . . .
```

Get and Set Element in List

```
import java.util.*;
public class ListExample2{
 public static void main(String args[]){
 //Creating a List
 List<String> list=new ArrayList<String>();
 //Adding elements in the List
 list.add("Mango");
 list.add("Apple");
 list.add("Banana");
 list.add("Grapes");
 //accessing the element
 System.out.println("Returning element: "+list.get(1));
 //changing the element
 list.set(1,"Dates");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);

 }
}
```

```
Returning element: Apple
Mango
Dates
Banana
Grapes
Press any key to continue . . .
```

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class ListExample5 {
public static void main(String[] args) {
    //Creating list of Books
    List<Book> list=new ArrayList<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to list
    list.add(b1);
    list.add(b2);
    list.add(b3);
    //Traversing list
    for(Book b:list){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Java ArrayList

- Java ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package.
- The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of the List interface here.

Java ArrayList

- The important points about the Java ArrayList class are:
- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:
- ArrayList<int> al = ArrayList<int>(); // does not work
- ArrayList<Integer> al = new ArrayList<Integer>(); // works fine

Methods of ArrayList

void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.

int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.

boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of the specified comparator.
Spliterator<E> spliterator()	It is used to create a spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements that lies within the given range.

Methods of ArrayList

int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Java ArrayList Example

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
       list.add("Mango");//Adding object in arraylist
       list.add("Apple");
       list.add("Banana");
       list.add("Grapes");
       //Printing the arraylist object
       System.out.println(list);
 }
}
```

Iterating ArrayList using Iterator

```
import java.util.*;
public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //Traversing list through Iterator
  Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and move to next
  }
 }
}
```

Java LinkedList class

- Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure.
- **The important points about Java LinkedList are:**
- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Methods of Java LinkedList

Method	Description
boolean add(E e)	It is used to append the specified element to the end of a list.
void add(int index, E element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
void clear()	It is used to remove all the elements from a list.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It is used to return true if a list contains a specified element.
Iterator<E> descendingIterator()	It is used to return an iterator over the elements in a deque in reverse sequential order.

E element()	It is used to retrieve the first element of a list.
E get(int index)	It is used to return the element at the specified position in a list.
E getFirst()	It is used to return the first element in a list.
E getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.
ListIterator<E> listIterator(int index)	It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list.
boolean offer(E e)	It adds the specified element as the last element of a list.
boolean offerFirst(E e)	It inserts the specified element at the front of a list.
boolean offerLast(E e)	It inserts the specified element at the end of a list.
E peek()	It retrieves the first element of a list

E peekFirst()	It retrieves the first element of a list or returns null if a list is empty.
E peekLast()	It retrieves the last element of a list or returns null if a list is empty.
E poll()	It retrieves and removes the first element of a list.
E pollFirst()	It retrieves and removes the first element of a list, or returns null if a list is empty.
E pollLast()	It retrieves and removes the last element of a list, or returns null if a list is empty.
E pop()	It pops an element from the stack represented by a list.
void push(E e)	It pushes an element onto the stack represented by a list.
E remove()	It is used to retrieve and removes the first element of a list.
E remove(int index)	It is used to remove the element at the specified position in a list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.
E removeFirst()	It removes and returns the first element from a list.
boolean removeFirstOccurrence(Object o)	It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail).
E removeLast()	It removes and returns the last element from a list.

Methods of Java LinkedList

boolean removeLastOccurrence(Object o)	It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).
E set(int index, E element)	It replaces the element at the specified position in a list with the specified element.
Object[] toArray()	It is used to return an array containing all the elements in a list in proper sequence (from first to the last element).
<T> T[] toArray(T[] a)	It returns an array containing all the elements in the proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array.
int size()	It is used to return the number of elements in a list.

Java LinkedList Example

```
import java.util.*;
public class LinkedList1{
 public static void main(String args[]){

  LinkedList<String> al=new LinkedList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
import java.util.*;
public class LinkedList2{
 public static void main(String args[]){
 LinkedList<String> ll=new LinkedList<String>();
           System.out.println("Initial list of elements: "+ll);
           ll.add("Ravi");
           ll.add("Vijay");
           ll.add("Ajay");
           System.out.println("After invoking add(E e) method: "+ll);
           //Adding an element at the specific position
           ll.add(1, "Gaurav");
           System.out.println("After invoking add(int index, E element) method: "+ll);
           LinkedList<String> ll2=new LinkedList<String>();
           ll2.add("Sonoo");
           ll2.add("Hanumat");
           //Adding second list elements to the first list
           ll.addAll(ll2);
           System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
           LinkedList<String> ll3=new LinkedList<String>();
           ll3.add("John");
           ll3.add("Rahul");
           //Adding second list elements to the first list at specific position
           ll.addAll(1, ll3);
           System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+ll);
           //Adding an element at the first position
           ll.addFirst("Lokesh");
           System.out.println("After invoking addFirst(E e) method: "+ll);
           //Adding an element at the last position
           ll.addLast("Harsh");
           System.out.println("After invoking addLast(E e) method: "+ll);

 }
}
```

Parul® University

Java LinkedList Example

```
Initial list of elements: []
After invoking add(E e) method: [Ravi, Vijay, Ajay]
After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]
After invoking addAll(Collection<? extends E> c) method: [Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addAll(int index, Collection<? extends E> c) method: [Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addFirst(E e) method: [Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addLast(E e) method: [Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]
Press any key to continue . . .
```

Java Deque interface

Parul®
University

- Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for "double ended queue".

Methods of Java Deque Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this deque and return true upon success.
boolean offer(object)	It is used to insert the specified element into this deque.
Object remove()	It is used to retrieves and removes the head of this deque.
Object poll()	It is used to retrieves and removes the head of this deque, or returns null if this deque is empty.
Object element()	It is used to retrieves, but does not remove, the head of this deque.
Object peek()	It is used to retrieves, but does not remove, the head of this deque, or returns null if this deque is empty.

ArrayDeque class

Parul® University

- The ArrayDeque class provides the facility of using deque and resizable-array.

- The important points about ArrayDeque class are:
- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

Java ArrayDeque Example

```
import java.util.*;
public class ArrayDequeExample {
    public static void main(String[] args) {
    //Creating Deque and adding elements
    Deque<String> deque = new ArrayDeque<String>();
    deque.add("Ravi");
    deque.add("Vijay");
    deque.add("Ajay");
    //Traversing elements
    for (String str : deque) {
    System.out.println(str);
    }
    }
}
```

Java ArrayDeque Example

```
import java.util.*;
public class DequeExample {
public static void main(String[] args) {
    Deque<String> deque=new ArrayDeque<String>();
    deque.offer("arvind");
    deque.offer("vimal");
    deque.add("mukul");
    deque.offerFirst("jai");
    System.out.println("After offerFirst Traversal...");
    for(String s:deque){
        System.out.println(s);
    }
    //deque.poll();
    //deque.pollFirst();//it is same as poll()
    deque.pollLast();
    System.out.println("After pollLast() Traversal...");
    for(String s:deque){
        System.out.println(s);
    }
}
}
```

```
After offerFirst Traversal...
jai
arvind
vimal
mukul
After pollLast() Traversal...
jai
arvind
vimal
```

Java HashSet

- Java HashSet class is used to create a collection that uses a hash table for storage. It implements Set interface.

- The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called hashing.
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet doesn't maintain the insertion order. Here, elements are
- inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.

Methods of Java HashSet class

SN	Modifier & Type	Method	Description
1)	boolean	add(E e)	It is used to add the specified element to this set if it is not already present.
2)	void	clear()	It is used to remove all of the elements from the set.
3)	object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

Java HashSet Example

Parul® University

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet();
              set.add("One");
              set.add("Two");
              set.add("Three");
              set.add("Four");
              set.add("Five");
              Iterator<String> i=set.iterator();
              while(i.hasNext())
              {
              System.out.println(i.next());
              }
 }
}
```

Java HashSet Example

Parul® University

```
import java.util.*;
class HashSet3{
 public static void main(String args[]){
  HashSet<String> set=new HashSet<String>();
            set.add("Ravi");
            set.add("Vijay");
            set.add("Arun");
            set.add("Sumit");
            System.out.println("An initial list of elements: "+set);
            //Removing specific element from HashSet
            set.remove("Ravi");
            System.out.println("After invoking remove(object) method: "+set);

            set.removeIf(str->str.contains("Vijay"));
            System.out.println("After invoking removeIf() method: "+set);
            //Removing all the elements available in the set
            set.clear();
            System.out.println("After invoking clear() method: "+set);
 }
}
```

```
An initial list of elements: [Vijay, Ravi, Arun, Sumit]
After invoking remove(object) method: [Vijay, Arun, Sumit]
After invoking removeIf() method: [Arun, Sumit]
After invoking clear() method: []
Press any key to continue . . .
```

Java TreeSet class

Parul® University

- The important points about Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

Methods of Java TreeSet class

Method	Description
boolean add(E e)	It is used to add the specified element to this set if it is not already present.
boolean addAll(Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.
E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Comparator<? super E> comparator()	It returns comparator that arranged elements in order.
Iterator descendingIterator()	It is used iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
E floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet headSet(E toElement)	It returns the group of elements that are less than the specified element.

NavigableSet headSet(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator iterator()	It is used to iterate the elements in ascending order.
E lower(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
E pollFirst()	It is used to retrieve and remove the lowest(first) element.
E pollLast()	It is used to retrieve and remove the highest(last) element.
Spliterator spliterator()	It is used to create a late-binding and fail-fast spliterator over the elements.
NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	It returns a set of elements that lie between the given range.

SortedSet subSet(E fromElement, E toElement))	It returns a set of elements that lie between the given range which includes fromElement and excludes toElement.
SortedSet tailSet(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet tailSet(E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
boolean contains(Object o)	It returns true if this set contains the specified element.
boolean isEmpty()	It returns true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void clear()	It is used to remove all of the elements from this set.
Object clone()	It returns a shallow copy of this TreeSet instance.
E first()	It returns the first (lowest) element currently in this sorted set.
E last()	It returns the last (highest) element currently in this sorted set.
int size()	It returns the number of elements in this set.

Java TreeSet Examples

Parul® University

```
import java.util.*;
class TreeSet1{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Java TreeSet Examples

Parul® University

```
import java.util.*;
class TreeSet3{
 public static void main(String args[]){
 TreeSet<Integer> set=new TreeSet<Integer>();
          set.add(24);
          set.add(66);
          set.add(12);
          set.add(15);
          System.out.println("Highest Value: "+set.pollFirst());
          System.out.println("Lowest Value: "+set.pollLast());
 }
}
```

```
Highest Value: 12
Lowest Value: 66
```

```
import java.util.*;
class TreeSet4{
 public static void main(String args[]){
  TreeSet<String> set=new TreeSet<String>();
          set.add("A");
          set.add("B");
          set.add("C");
          set.add("D");
          set.add("E");
          System.out.println("Initial Set: "+set);

          System.out.println("Reverse Set: "+set.descendingSet());

          System.out.println("Head Set: "+set.headSet("C", true));

          System.out.println("SubSet: "+set.subSet("A", false, "E", true));

          System.out.println("TailSet: "+set.tailSet("C", false));
 }
}
```

```
Initial Set: [A, B, C, D, E]
Reverse Set: [E, D, C, B, A]
Head Set: [A, B, C]
SubSet: [B, C, D, E]
TailSet: [D, E]
```

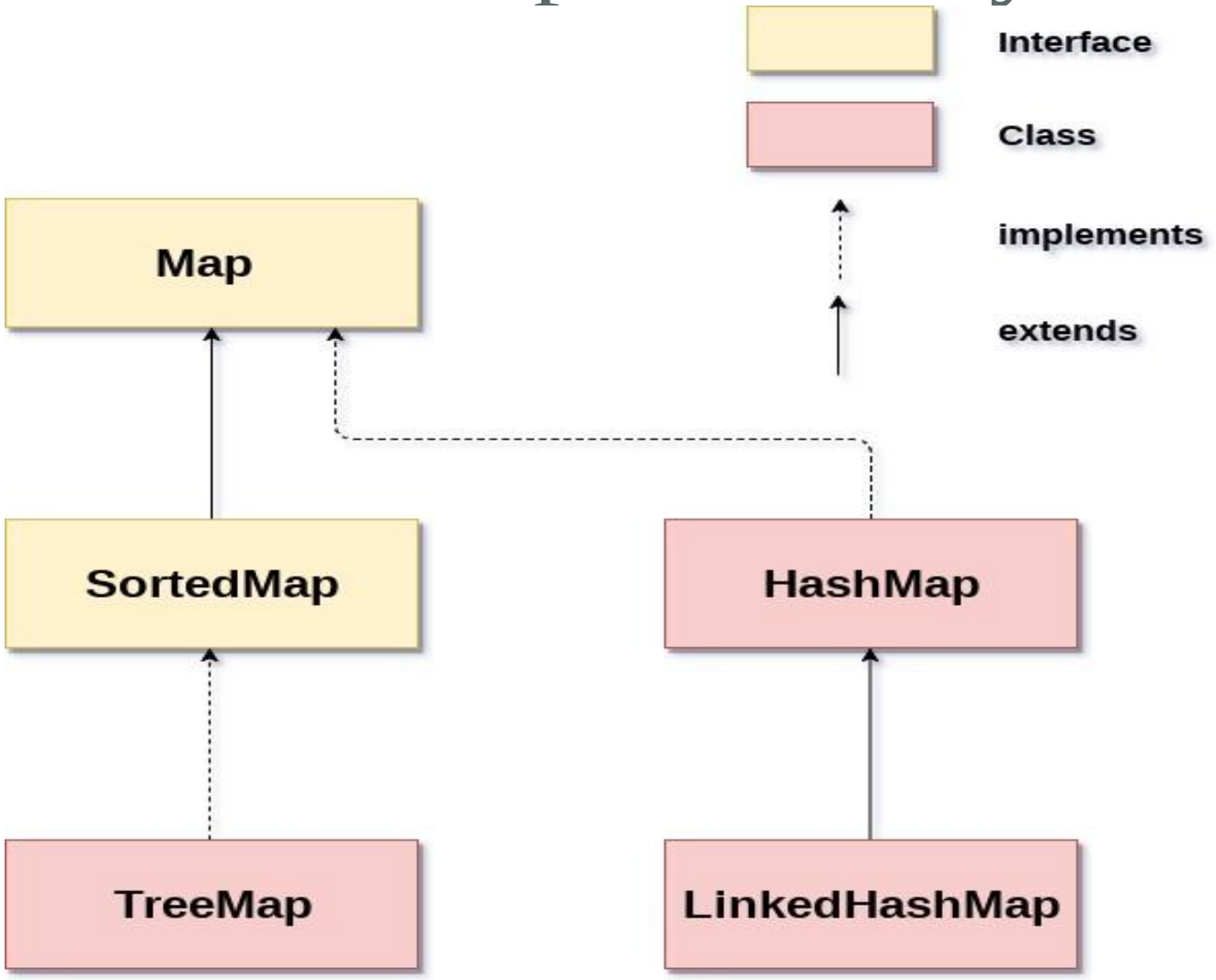
Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

Java Map Hierarchy



Methods of Map interface

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
Set<Map.Entry<K,V>> entrySet()	It returns the Set view containing all the keys and values.
void clear()	It is used to reset the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).

V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.

Methods of Map interface

V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Java Map Example

```
import java.util.*;
class MapExample2{
 public static void main(String args[]){
  Map<Integer,String> map=new HashMap<Integer,String>();
  map.put(100,"Amit");
  map.put(101,"Vijay");
  map.put(102,"Rahul");
  //Elements can traverse in any order
  for(Map.Entry m:map.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Java HashMap Class

- Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

- It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value.

Java HashMap Class

- Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap maintains no order.

Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.

boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.

V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection<V> values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Java HashMap Example

```
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(4,"Grapes");

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

```
Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes
```

No Duplicate Key on HashMap

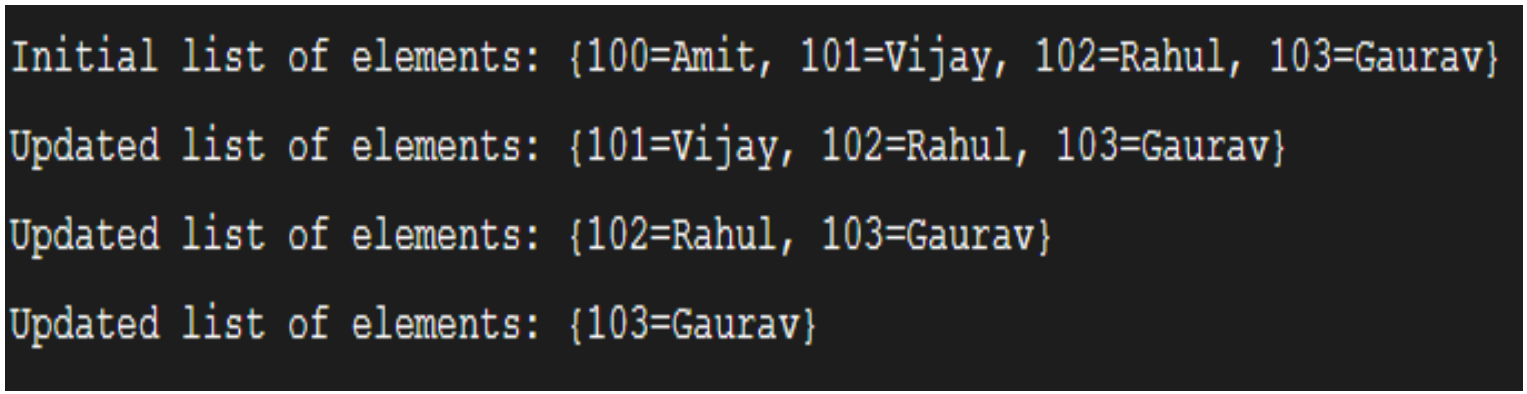
```
import java.util.*;
public class HashMapExample2{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(1,"Grapes"); //trying duplicate key

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

```
Iterating Hashmap...
1 Grapes
2 Apple
3 Banana
```

Java HashMap example to remove() elements


```
import java.util.*;
public class HashMap2 {
    public static void main(String args[]) {
     HashMap<Integer,String> map=new HashMap<Integer,String>();
       map.put(100,"Amit");
       map.put(101,"Vijay");
       map.put(102,"Rahul");
       map.put(103, "Gaurav");
     System.out.println("Initial list of elements: "+map);
     //key-based removal
     map.remove(100);
     System.out.println("Updated list of elements: "+map);
     //value-based removal
     map.remove(101);
     System.out.println("Updated list of elements: "+map);
     //key-value pair based removal
     map.remove(102, "Rahul");
     System.out.println("Updated list of elements: "+map);
    }
}
```

Java HashMap example to remove() elements



1. Collections Framework Overview

The Collections Framework organizes and standardizes how collections (groups of objects) are handled in Java. It provides:

- **Interfaces**: Abstract data types representing collections (e.g., List, Set).
- **Implementations**: Concrete classes (e.g., ArrayList, HashSet) that implement these interfaces.
- **Algorithms**: Utility methods (in the Collections class) for sorting, searching, and manipulating collections.

2. Collections Class

The Collections class is a utility class that provides static methods to manipulate collections like sorting, searching, and thread-safety enhancements.

Common Methods in Collections Class

1 **Sorting**:

Collections.sort(list);

2 **Shuffling**:

Collections.shuffle(list);

3 **Reverse**

Collections.reverse(list);

4 **Synchronization**

3. List Interface

A List represents an ordered collection of elements. It allows duplicates and provides positional access to elements.

Key Implementations

1. **ArrayList**: A resizable array. Best for fast random access.

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

list.add("Java");

list.add("Python");

System.out.println(list); // [Java, Python]
```

2 **LinkedList**: A doubly-linked list. Best for frequent insertions and deletions.

```
List<String> linkedList = new LinkedList<>();
linkedList.add("C++");
```

```
linkedList.add("Ruby");
System.out.println(linkedList); // [C++, Ruby]
```

3. **Vector**: A synchronized version of ArrayList. Rarely used today.

---

3. List Interface

A List represents an ordered collection of elements. It allows duplicates and provides positional access to elements.

Key Implementations

1. **ArrayList**: A resizable array. Best for fast random access.

java

Copy code

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

list.add("Java");

list.add("Python");

System.out.println(list); // [Java, Python]
```

2. **LinkedList**: A doubly-linked list. Best for frequent insertions and deletions.

java

Copy code

```
List<String> linkedList = new LinkedList<>();

linkedList.add("C++");

linkedList.add("Ruby");

System.out.println(linkedList); // [C++, Ruby]
```

3. **Vector**: A synchronized version of ArrayList. Rarely used today.

---

4. Set Interface

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

Key Implementations

1. **HashSet**: Backed by a hash table; does not maintain order.

```
Set<String> set = new HashSet<>();
```

```
set.add("Apple");
set.add("Banana");
set.add("Apple"); // Duplicate, won't be added
System.out.println(set); // [Apple, Banana]
```

2 LinkedHashSet: Maintains insertion order.

```
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("One");
linkedHashSet.add("Two");
System.out.println(linkedHashSet); // [One, Two]
```

3 **TreeSet**: Sorted in natural order.

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

5. Enumeration

The Enumeration interface provides a way to traverse through legacy collections like Vector or Hashtable.

```
Vector<String> vector = new Vector<>();
vector.add("A");
vector.add("B");

Enumeration<String> enumeration = vector.elements();
while (enumeration.hasMoreElements()) {
   System.out.println(enumeration.nextElement());
}
```

---

6. Iterator

The Iterator interface provides a modern way to traverse elements in a collection. It replaces Enumeration.

Key Methods

- hasNext(): Checks if more elements are available.
- next(): Retrieves the next element.
- remove(): Removes the last retrieved element.

Usage

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

list.add("X");

list.add("Y");


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

while (iterator.hasNext()) {

   System.out.println(iterator.next());

}
```

The **Java Collection Framework** provides a set of classes and interfaces for managing and manipulating groups of objects. It includes data structures like lists, sets, queues, and maps, along with utilities to manipulate them. Here's a breakdown of key components related to the topics you've mentioned:

---

1. Collections Framework Overview

The Collections Framework organizes and standardizes how collections (groups of objects) are handled in Java. It provides:

- **Interfaces**: Abstract data types representing collections (e.g., List, Set).
- **Implementations**: Concrete classes (e.g., ArrayList, HashSet) that implement these interfaces.
- **Algorithms**: Utility methods (in the Collections class) for sorting, searching, and manipulating collections.

---

2. Collections Class

The Collections class is a utility class that provides static methods to manipulate collections like sorting, searching, and thread-safety enhancements.

Common Methods in Collections Class

1. **Sorting**:

java

Copy code

```
Collections.sort(list);
```

2. **Shuffling**:

java

Copy code

```
Collections.shuffle(list);
```

3. **Reverse**:

java

Copy code

```
Collections.reverse(list);
```

4. **Synchronization**:

java

Copy code

```
List<String> synchronizedList = Collections.synchronizedList(list);
```

---

3. List Interface

A List represents an ordered collection of elements. It allows duplicates and provides positional access to elements.

Key Implementations

1. **ArrayList**: A resizable array. Best for fast random access.

java

Copy code

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

```
list.add("Java");
list.add("Python");
System.out.println(list); // [Java, Python]
```

2. **LinkedList**: A doubly-linked list. Best for frequent insertions and deletions.

```
java
Copy code
List<String> linkedList = new LinkedList<>();
linkedList.add("C++");
linkedList.add("Ruby");
System.out.println(linkedList); // [C++, Ruby]
```

3. **Vector**: A synchronized version of ArrayList. Rarely used today.

---

4. Set Interface

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

Key Implementations

1. **HashSet**: Backed by a hash table; does not maintain order.

```
java
Copy code
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple"); // Duplicate, won't be added
System.out.println(set); // [Apple, Banana]
```

2. **LinkedHashSet**: Maintains insertion order.

```
java
Copy code
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("One");
linkedHashSet.add("Two");
```

System.out.println(linkedHashSet); // [One, Two]

3. **TreeSet**: Sorted in natural order.

java

Copy code

Set<Integer> treeSet = new TreeSet<>();

treeSet.add(3);

treeSet.add(1);

treeSet.add(2);

System.out.println(treeSet); // [1, 2, 3]

---

5. Enumeration

The Enumeration interface provides a way to traverse through legacy collections like Vector or Hashtable.

Usage

java

Copy code

Vector<String> vector = new Vector<>();

vector.add("A");

vector.add("B");

Enumeration<String> enumeration = vector.elements();

while (enumeration.hasMoreElements()) {

System.out.println(enumeration.nextElement());

}

---

6. Iterator

The Iterator interface provides a modern way to traverse elements in a collection. It replaces Enumeration.

Key Methods

- hasNext(): Checks if more elements are available.

- next(): Retrieves the next element.
- remove(): Removes the last retrieved element.

Usage

java

Copy code

```
List<String> list = new ArrayList<>();
list.add("X");
list.add("Y");

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

---

7. ArrayList

The ArrayList is a resizable array implementation of the List interface.

Key Features

1. Allows duplicate elements.
2. Maintains insertion order.
3. Random access using indexes.
4. Automatically resizes when full.

Usage

```
import java.util.ArrayList;

public class ArrayListExample {
   public static void main(String[] args) {
      ArrayList<String> arrayList = new ArrayList<>();
      arrayList.add("Hello");
      arrayList.add("World");
```

```
        System.out.println(arrayList); // [Hello, World]

        // Accessing elements
        System.out.println(arrayList.get(1)); // World

        // Removing an element
        arrayList.remove(0);
        System.out.println(arrayList); // [World]
    }
}
```

Summary of Interfaces and Implementations

Interface	Key Implementations	Features
`List`	`ArrayList`, `LinkedList`	Ordered, allows duplicates.
`Set`	`HashSet`, `LinkedHashSet`, `TreeSet`	Unordered or ordered, no duplicates.
`Iterator`	NA	Allows traversing modern collections.
`Enumeration`	NA	Legacy traversal for older collections like `Vector`.