Day 6- Facilitation Guide

Index

For (1.5 hrs) ILT

In the previous session on exception handling in Java, we discussed several key concepts:

Types of Exceptions: Java exceptions are categorized into two main types: checked exceptions and unchecked exceptions (runtime exceptions). Checked exceptions are checked by the compiler at compile-time and must be either caught using try/catch or declared using the throws clause. Unchecked exceptions are not checked at compile-time, and the compiler does not enforce their handling or declaration.

try/catch Block: The try/catch block is used to handle exceptions. Code that might throw exceptions is enclosed within the try block, and potential exceptions are caught and handled in the catch block. Multiple catch blocks can be used to handle different types of exceptions.

throws Clause: The throws clause is used in method declarations to specify the exceptions that the method might throw. It helps in informing the calling code about potential exceptions that need to be managed.

Exception Hierarchy: Java exceptions are organized in a hierarchy. All exceptions inherit from the base class Java.lang.Exception. Unchecked exceptions inherit from java.lang.RuntimeException. This hierarchy aids in structuring exception handling code effectively.

Checked vs. Unchecked Exceptions: Checked exceptions are typically used for recoverable scenarios, where the application can handle the exceptional situation. Unchecked exceptions are often caused by programming errors and might indicate issues that are difficult to recover from.

Best Practices: It's important to follow best practices in exception handling, such as handling exceptions at an appropriate level of abstraction, not catching exceptions without a clear reason, and providing meaningful error messages.

try-with-resources: Java provides the try-with-resources statement to automatically manage resources like files or connections that need to be closed after usage. This improves resource management and reduces the chance of resource leaks.

Custom Exceptions: Developers can create custom exception classes by extending the existing exception classes to handle specific application-related exceptional scenarios.

By understanding these concepts and applying them in your code, you can create more robust and reliable Java applications that handle exceptions effectively, leading to better error management and user experience.

During this session, we'll delve into the Collection Framework and its various implementation classes.

Food for thought…

Trainer should ask the students Do you find yourself wondering how Java handles scenarios where you need to store a bunch of elements and perform various operations on them? The Collection Framework might just hold the answers you seek.

I. Generics and Collections

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the Java programmer to store a specific type of objects.

Without generics in Java's Collection Framework, several issues can arise, primarily related to type safety, readability, and maintainability of code. Some of the common problems include:

Lack of Type Safety: Without generics, collections can store objects of any type. This can lead to runtime errors if you inadvertently store objects of the wrong type and attempt to retrieve them without proper casting.

Type Casting Issues: Retrieving elements from collections without generics requires manual type casting, which can lead to errors if not done correctly. Incorrect casting can result in ClassCastException at runtime.

Reduced Readability: Code without generics might not clearly convey the intended types of data stored in collections, making it harder to understand the purpose and behavior of the code.

Limited Compile-Time Checks: Since type information is not enforced at compile-time, issues related to incorrect types might only surface at runtime during execution, leading to unexpected behavior and bugs.

Type Confusion: Mixing different types of objects in the same collection can lead to confusion and errors, especially when iterating over the collection and performing operations on the elements.

In summary, the absence of generics in the Collection Framework can lead to a range of problems related to type safety, code readability, maintainability, and overall robustness of the codebase. Generics were introduced to address these issues and provide a more type-safe, efficient, and readable way to work with collections in Java.

Advantage of Java Generics

Generics in Java's Collection Framework offer several advantages that enhance type safety, code clarity, and reusability. Some of the key advantages include:

Type Safety: Generics ensure that the type of elements stored in a collection is known at compile-time. This eliminates the possibility of runtime type errors, as incorrect types are caught during compilation.

Compile-Time Checks: With generics, the compiler enforces type checks and can catch type-related errors early in the development process, reducing the likelihood of bugs and improving overall code quality.

Code Reusability: Generics allow you to write reusable code that can work with different data types. By using generic methods and classes, you can avoid duplicating similar code for different data types.

Elimination of Type Casting: Prior to generics, collections required explicit type casting when retrieving elements. Generics eliminate the need for such casting, resulting in cleaner and more concise code.

Enhanced Readability: Generic code is often more readable because it clearly states the intended types for variables and collections, making the code's purpose and behavior more apparent.

Compile-Time Type Inference: The diamond operator (<>) introduced in Java 7 allows the compiler to infer the generic type arguments, reducing the verbosity of code while maintaining type safety.

Reduced Chance of Bugs: By preventing inappropriate data types from entering collections, generics help in reducing the chances of logical errors and unexpected behaviors in the code.

Collections with Custom Types: Generics enable you to create collections that can hold custom classes, improving encapsulation and encapsulating specific logic within those classes.

In summary, generics in the Collection Framework provide a powerful mechanism for type-safe and reusable code, leading to improved code quality, reduced bugs, and enhanced readability. They enable developers to write more expressive and efficient code while ensuring a higher level of type correctness at compile-time.

II. Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

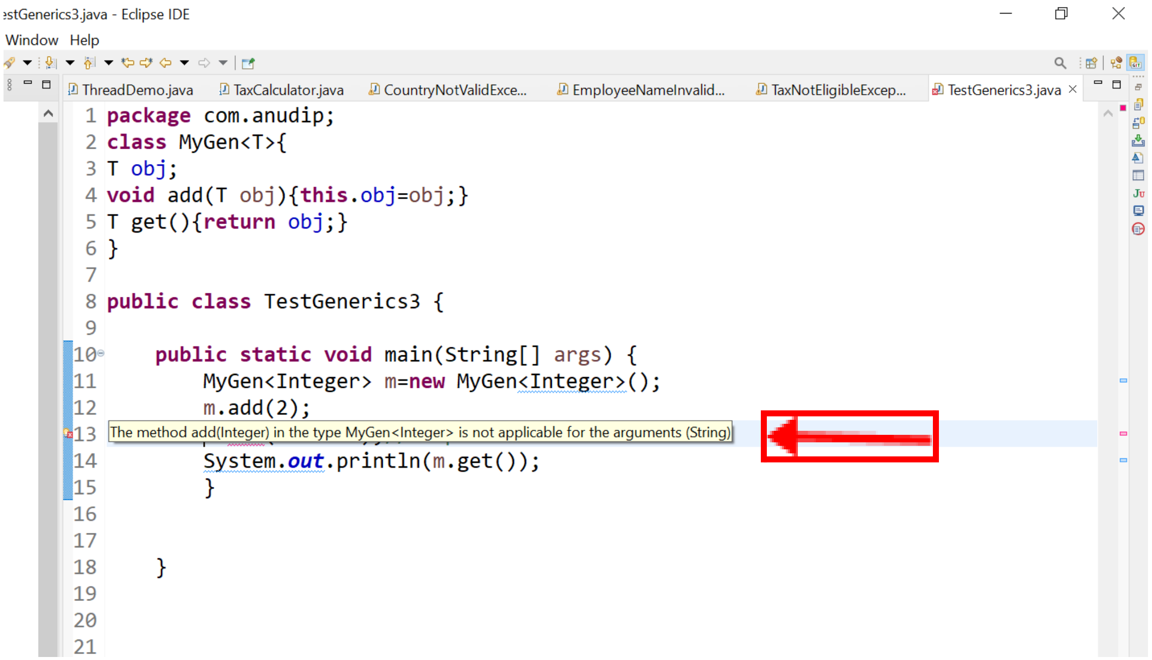
```
 class MyGen<T>{

T obj;

void add(T obj){this.obj=obj;}

T get(){return obj;}

}
```

The **T** type indicates that it can refer to any type (like String, Integer, and Employee).

The type you specify for the class will be used to store and retrieve the data.

```
class TestGenerics3{
```

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

MyGen<Integer> m=new MyGen<Integer>();

m.add(2);

//m.add("vivek"); //Compile time error as tye type specified while creating object is Integer

System.out.println(m.get());

}}
```



```
package com.anudip;
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}

public class TestGenerics3 {

    public static void main(String[] args) {
        MyGen<Integer> m=new MyGen<Integer>();
        m.add(2);

        System.out.println(m.get());
        }


    }
```

The method add(Integer) in the type MyGen<Integer> is not applicable for the arguments (String)

Type Parameters

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

What is Collection FrameWork

Collection Framework offers the capability to Java to represent a group of elements in classes and Interfaces.

Collection Framework enables the user to perform various data manipulation operations like storing data, searching, sorting, insertion, deletion, and updating of data on the group of elements. Followed by the Java Collections Framework, you must learn and understand the hierarchy of Java collections and various descendants or classes and interfaces involved in the Java Collections.

What is a framework

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

A Collection Framework provides an architecture to store and manipulate a group of objects.Collection Framework includes the following:

- Interfaces
- Classes
- Algorithm

Let's learn about them in detail:

Interfaces: Interface in Java refers to the abstract data types. They allow collections to be manipulated independently from the details of their representation. Also, they form a hierarchy in object-oriented programming languages.

Classes: Classes in Java are the implementation of the collection interface. It basically refers to the data structures that are used again and again.

Algorithm: Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces. Algorithms are polymorphic in nature as the same method can be used to take many forms or you can say perform different implementations of the Java Collection interface.

***Question:* So why do you think we need Collection Framework?**

The Collection Framework provides the developers to access prepackaged data structures as well as algorithms to manipulate data.

***Question:*What is the purpose of using a Collection?**

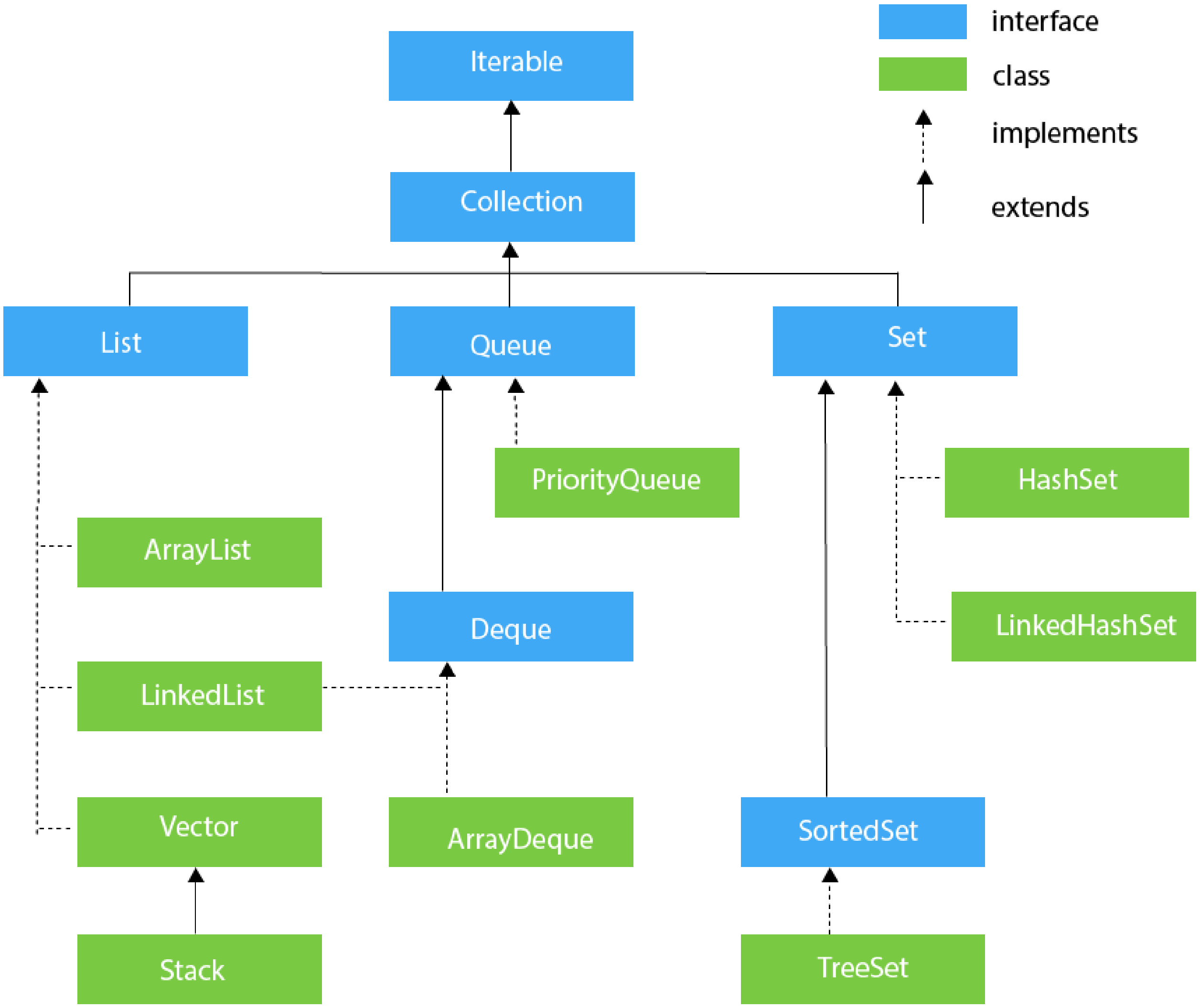
There are several benefits of using Collection such as:

- Reducing the effort required to write the code by providing useful data structures and algorithms
- Collections provide high-performance and high-quality data structures and algorithms thereby increasing the speed and quality
- Unrelated APIs can pass collection interfaces back and forth
- Decreases extra effort required to learn, use, and design new API's
- Supports reusability of standard data structures and algorithms

Next, let us move to the Java Collections Framework hierarchy and see where these interfaces and classes reside.

Collection Framework Hierarchy

As we have learned, the Java Collection Framework includes interfaces and classes. Now, let us see the hierarchy.



Iterator interface

The Iterator interface in Java is a fundamental part of the Java Collections Framework. It provides a way to traverse (or iterate over) the elements of a collection in a consistent and efficient manner, regardless of the specific collection implementation.

Methods of Iterator interface:

There are only three methods in the Iterator interface. They are:

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.

Note:*An example is provided subsequently.*

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 serves as the foundation for classes that represent a collection of elements and allows them to be iterated over using the enhanced for loop and provides a way to obtain an Iterator for traversing the elements.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

Collection Interface

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.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

III. List Interface

The List interface is a core part of the Java Collections Framework and represents an ordered collection of elements. It defines a contract for classes that implement lists, allowing elements to be stored in a specific order and accessed by their indices.

Here are key points about the List interface:

Ordered Collection: A List maintains the order in which elements are inserted. You can access elements by their index, starting from 0.

Duplicates Allowed: Unlike some other collection types, a List can contain duplicate elements.

Methods: The List interface extends the Collection interface and adds methods specific to lists, including:

add(E element): Adds an element to the end of the list.
add(int index, E element): Inserts an element at the specified index.
get(int index): Retrieves the element at the specified index.
set(int index, E element): Replaces the element at the specified index.
remove(int index): Removes the element at the specified index.
indexOf(Object o): Returns the index of the first occurrence of the specified element.
lastIndexOf(Object o): Returns the index of the last occurrence of the specified element.
subList(int fromIndex, int toIndex): Returns a view of the list between the specified indices.
Common Implementations: Some common implementations of the List interface are ArrayList (dynamic array), LinkedList (doubly-linked list), and Vector (similar to ArrayList but synchronized).

Iterating Over a List: You can iterate over a List using the enhanced for loop, the traditional for loop with indices, or by obtaining an iterator using the iterator() method.

Null Elements: A List can contain null elements unless explicitly disallowed by a specific implementation.

Manipulation and Reordering: List methods allow for adding, removing, and reordering elements, enabling you to modify the list's content dynamically.

Sorting: The Collections class provides a utility method sort() to sort the elements of a List.

Sublists: The subList() method allows you to extract a portion of the list as a separate sublist.

Index Validity: Keep in mind that methods that involve indices may throw IndexOutOfBoundsException if the index is out of range.

The List interface provides a versatile way to work with ordered collections of elements, making it suitable for scenarios where maintaining element order and accessing elements by index are important requirements.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

```
List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

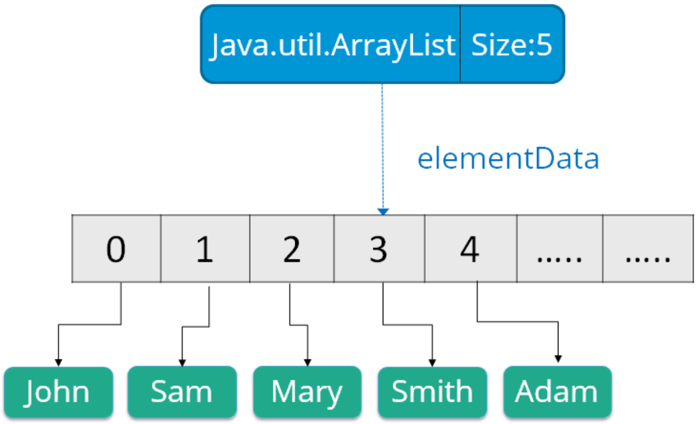
ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

***Question:**Why is ArrayList better than Array?*

The limitation with array is that it has a fixed length so if it is full you cannot add any more elements to it, likewise if there are a number of elements removed from it, the memory consumption would be the same as it doesn’t shrink.

On the other hand, ArrayList **can dynamically grow and shrink** after addition and removal of elements. Apart from these benefits, the ArrayList class enables us to use predefined methods which makes our task easy.



Syntax:

List <data-type> list1= new ArrayList();

Or

ArrayList<data-type> list1=new ArrayList<data-type>();

Some of the methods in array list are listed below:

1) **add(Object o):** This method adds an object o to the arraylist.

obj.add("hello");

This statement would add a string hello in the arraylist at last position.

2) **add(int index, Object o):** It adds the object o to the array list at the given index.

obj.add(2, "bye");

It will add the string bye to the 2nd index (3rd position as the array list starts with index 0) of the array list.

3) **remove(Object o):** Removes the object o from the ArrayList.

obj.remove("Chaitanya");

This statement will remove the string “Chaitanya” from the ArrayList.

4) **remove(int index):** Removes element from a given index.

obj.remove(3);

It would remove the element of index 3 (4th element of the list – List starts with o).

5) **set(int index, Object o):** Used for updating an element. It replaces the element present at the specified index with the object o.

```
obj.set(2, "Tom");
```

It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

6) **int indexOf(Object o):** Gives the index of the object o. If the element is not found in the list then this method returns the value -1.

```
int pos = obj.indexOf("Tom");
```

This would give the index (position) of the string Tom in the list.

7) **Object get(int index):** It returns the object of the list which is present at the specified index.

```
String str= obj.get(2);
```

Function get would return the string stored at 3rd position (index 2) and would be assigned to the string “str”. We have stored the returned value in the string variable because in our example we have defined the ArrayList is of String type. If you are having an integer array list then the returned value should be stored in an integer variable.

8) **int size():** It gives the size of the ArrayList – Number of elements of the list.

```
int numberOfItems = obj.size();
```

9) **boolean contains(Object o):** It checks whether the given object o is present in the array list if it's there then it returns true else it returns false.

```
obj.contains("Steve");
```

It would return true if the string “Steve” is present in the list else we would get false.

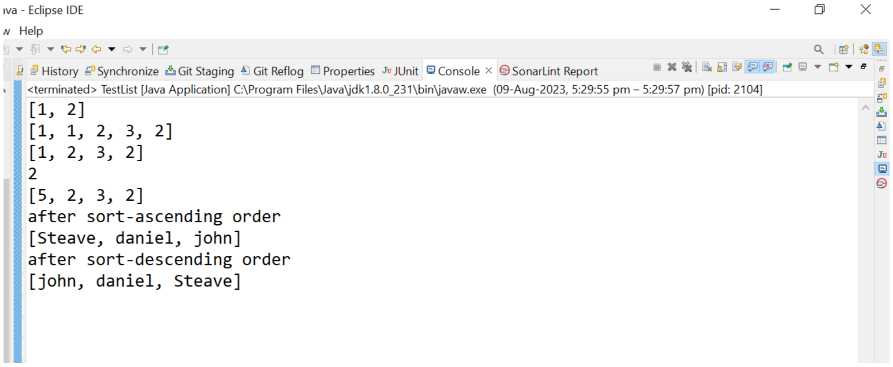
10) **clear():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

```
obj.clear();
```

// Java program to Demonstrate List Interface

```
// Importing all utility classes
import java.util.*;

// Main class
// ListDemo class
class Test{

    // Main driver method
    public static void main(String[] args)
    {

        // Creating an object of List interface
        // implemented by the ArrayList class
        List<Integer> l1 = new ArrayList<Integer>();

        // Adding elements to object of List interface
        // Custom inputs

        l1.add(0, 1);
        l1.add(1, 2);

        // Print the elements inside the object
        System.out.println(l1);

        // Now creating another object of the List
        // interface implemented ArrayList class
        // Declaring object of integer type
        List<Integer> l2 = new ArrayList<Integer>();

        // Again adding elements to object of List interface
        // Custom inputs
        l2.add(1);
        l2.add(2);
        l2.add(3);

        // Will add list l2 from 1 index
        l1.addAll(1, l2);

        System.out.println(l1);
        // Removes element from index 1
        l1.remove(1);

        // Printing the updated List 1
        System.out.println(l1);

        // Prints element at index 3 in list 1
        // using get() method
        System.out.println(l1.get(3));

        // Replace 0th element with 5
        // in List 1
        l1.set(0, 5);

        // Again printing the updated List 1
        System.out.println(l1);

                 //sort the arraylist
                 List<String> listName = new ArrayList<String>();
                listName.add(“john”);
                listName.add(“Steave”);
                 listName.add(“daniel”);

                    Collections.sort(listName);
                    System.out.println("after sort-ascending order");
                    System.out.println(listName);

                    Collections.sort(listName, Collections.reverseOrder());
                    System.out.println("after sort-descending order");
                    System.out.println(listName);

    }
}
```

Output:



```
<terminated> TestList [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe  (09-Aug-2023, 5:29:55 pm – 5:29:57 pm) [pid: 2104]
[1, 2]
[1, 1, 2, 3, 2]
[1, 2, 3, 2]
2
[5, 2, 3, 2]
after sort-ascending order
[Steave, daniel, john]
after sort-descending order
[john, daniel, Steave]
```

In the above program the sort() method we used, comes from the Collections class. This makes us wonder: what exactly is the Collections class and how do we use it?

In Java, the Collections class is a utility class provided by the Java Collections Framework, which is a set of interfaces and classes that provide a unified and organized way to handle collections of objects. The Collections class contains various static methods that operate on or return collections, offering utility functionalities that are commonly needed when working with collections. It's part of the java.util package.

Here are some methods provided by Collections class for tasks such as:

Sorting: Methods like sort(List<T> list) to sort elements of a List in their natural order or using a custom comparator.

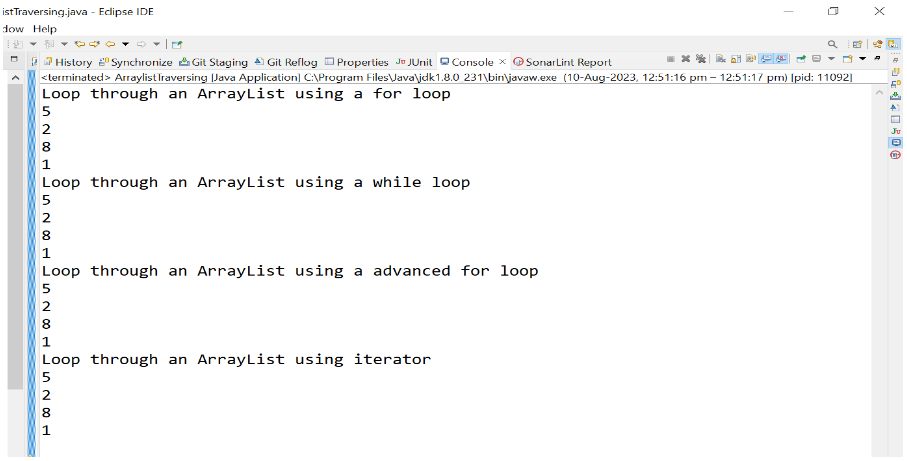
Searching: Methods like binarySearch(List<? extends Comparable<? super T>> list, T key) to perform binary search on a sorted List.

Min and Max: Methods like min(Collection<? extends T> coll) and max(Collection<? extends T> coll) to find the minimum and maximum elements in a Collection.

Iterate through an ArrayList using various loops and Iterator

```
package com.anudip;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class ArraylistTraversing {
        public static void main(String[] args) {
                List<Integer> numbers = new ArrayList<>();
        numbers.add(5);
        numbers.add(2);
        numbers.add(8);
        numbers.add(1);
        System.out.println("Loop through an ArrayList using a for loop");
        for(int i=0;i<numbers.size();i++)
        {
                System.out.println(numbers.get(i));
        }
        System.out.println("Loop through an ArrayList using a while loop");
        int i=0;
        while(numbers.size()>i)
        {
                System.out.println(numbers.get(i));
                i++;
        }
        System.out.println("Loop through an ArrayList using a "
                        + "advanced for loop");
        for(Integer in:numbers)
        {
                System.out.println(in);
        }

        System.out.println("Loop through an ArrayList using iterator");

        Iterator it=numbers.iterator();
        while(it.hasNext())
        {
                System.out.println(it.next());
        }
       }
}
```

Output:



```
<terminated> ArraylistTraversing [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe  (10-Aug-2023, 12:51:16 pm – 12:51:17 pm) [pid: 11092]
Loop through an ArrayList using a for loop
5
2
8
1
Loop through an ArrayList using a while loop
5
2
8
1
Loop through an ArrayList using a advanced for loop
5
2
8
1
Loop through an ArrayList using iterator
5
2
8
1
```

ArrayList of type custom object

An ArrayList of custom types in Java refers to an instance of the ArrayList class that holds objects of a user-defined or custom class. This allows you to create a dynamic list that contains instances of your own custom class, enabling you to work with collections of your specific data structures.

Here's an example of how you might create and use an ArrayList of a custom type:

Suppose you have a custom class named Employee:

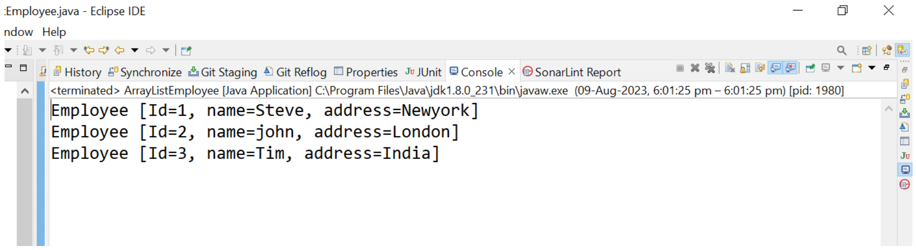
```
package collectionsdemo;
public class Employee {
private int Id;
private String name;
private String address;
public int getId() {
        return Id;
}
public void setId(int id) {
        Id = id;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getAddress() {
    return address;
}
public void setAddress(String address) {
    this.address = address;
}
public Employee(int id, String name, String address) {
    super();
    Id = id;
    this.name = name;
    this.address = address;
}
public Employee() {
    super();
    // TODO Auto-generated constructor stub
}
@Override
public String toString() {
    return "Employee [Id=" + Id + ", name=" + name + ", address=" + address + "]";
}
}
```

You can create an ArrayList that holds instances of the Employee class:

```
package collectionsdemo;

import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListEmployee {

    public static void main(String[] args) {
        ArrayList<Employee> employees=new ArrayList<Employee>();
        Scanner sc=new Scanner(System.in);

        employees.add(new Employee(1, "Steve", "Newyork"));
        employees.add(new Employee(2, "john", "London"));
        employees.add(new Employee(3, "Tim", "India"));

    for(Employee e:employees) {
```

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

        }

}
```

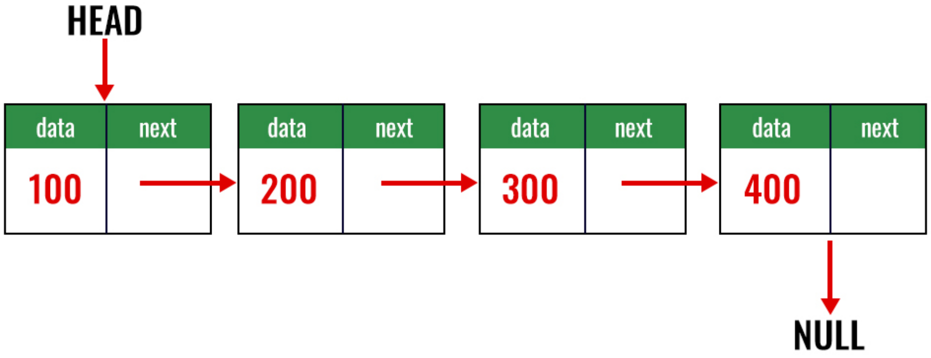
Output:



```
<terminated> ArrayListEmployee [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe  (09-Aug-2023, 6:01:25 pm – 6:01:25 pm) [pid: 1980]
Employee [Id=1, name=Steve, address=Newyork]
Employee [Id=2, name=john, address=London]
Employee [Id=3, name=Tim, address=India]
```

LinkedList in Java

Similar to arrays in Java, LinkedList is a linear data structure. However LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.



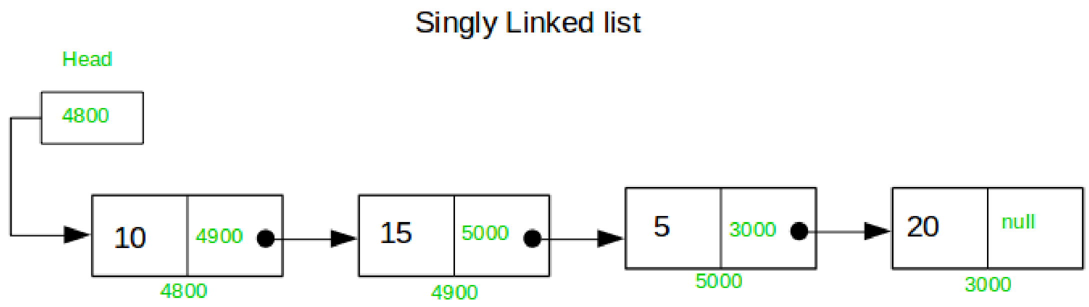
LinkedList representation

Each element in the LinkedList is called the Node. Each Node of the LinkedList contains two items: 1) Content of the element 2) Pointer/Address/Reference to the Next Node in the LinkedList.

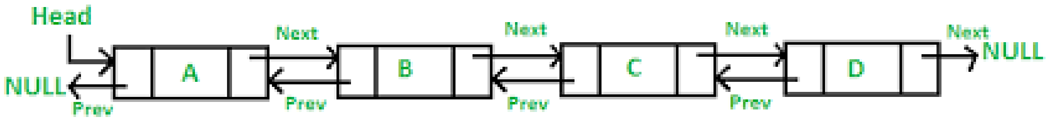
Java Linked List class uses two types of Linked list to store the elements:

- Singly Linked List
- Doubly Linked List

Singly Linked List: In a singly Linked list each node in this list stores the data of the node and a pointer or reference to the next node in the list. Refer to the below image to get a better understanding of single Linked list.



Doubly Linked List: In a doubly Linked list, it has two references, one to the next node and another to the previous node. You can refer to the below image to get a better understanding of doubly linked lists.



Note:

1. **Head** of the LinkedList only contains the Address of the **First element** of the List.
2. The Last element of the LinkedList contains **null** in the pointer part of the node because it is the end of the List so it doesn’t point to anything as shown in the above diagram.

Question:*Why do we need a Linked List?*

You must be aware of the arrays which is also a linear data structure but **arrays have certain limitations such as:**

1) **Size of the array is fixed** which is decided when we create an array so it is hard to predict the number of elements in advance, if the declared size fall short then we cannot increase the size of an array and if we declare a large size array and do not need to store that many elements then it is a waste of memory.

2) Array elements **need contiguous memory locations** to store their values.

3) **Inserting an element in an array is performance wise expensive** as we have to shift several elements to make a space for the new element. For example:
Let’s say we have an array that has the following elements: 10, 12, 15, 20, 4, 5, 100, now if we want to insert a new element 99 after the element that has value 12 then we have to shift all the elements after 12 to their right to make space for new element.

Similarly **deleting an element** from the array is also a performance wise expensive operation because all the elements after the deleted element have to be shifted left.

These limitations are handled in the Linked List by providing following features:

1. Linked list allows **dynamic memory allocation**, which means memory allocation is done at the run time by the compiler and we do not need to mention the size of the list during linked list declaration.

2. Linked list elements **don’t need contiguous memory locations** because elements are linked with each other using the reference part of the node that contains the address of the next node of the list.

3. Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list does not require element shifting, only the pointer of the previous and the next node requires change.

Methods of LinkedList class:

1) **boolean add(Object item)**: It adds the item at the end of the list.

```
llistObj.add("Hello");
```

It would add the string “Hello” at the end of the linked list.

2) **void add(int index, Object item)**: It adds an item at the given index of the list.

```
llistObj.add(2, "bye");
```

This will add the string “bye” at the 3rd position(2 index is 3rd position as index starts with 0).

3) **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws a NullPointerException if the specified collection is null. Consider the below example –

```
LinkedList<String> llistObj= new LinkedList<String>();

ArrayList<String> arraylist= new ArrayList<String>();

arraylist.add("String1");

arraylist.add("String2");

llistObj.addAll(arraylist);
```

This piece of code would add all the elements of ArrayList to the LinkedList.

4) **boolean addAll(int index, Collection c)**: It adds all the elements of collection c to the list starting from a given index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.

```
llistObj.add(5, arraylist);
```

It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).

5) **void addFirst(Object item)**: It adds the item (or element) at the first position in the list.

```
llistObj.addFirst("text");
```

It would add the string “text” at the beginning of the list.

6) **void addLast(Object item)**: It inserts the specified item at the end of the list.

```
llistObj.addLast("Chaitanya");
```

This statement will add a string “Chaitanya” at the end position of the linked list.

7) **void clear()**: It removes all the elements of a list.

```
llistObj.clear();
```

8) **Object clone()**: It returns a copy of the list.

For e.g. My linkedList has four items: text1, text2, text3 and text4.

```
Object str= llistObj.clone();

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

Output: The output of above code would be:

```
[text1, text2, text3, text4]
```

9) **boolean contains(Object item)**: It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

```
boolean var = llistObj.contains("TestString");
```

It will check whether the string “TestString” exist in the list or not.

10) **Object get(int index)**: It returns the item of the specified index from the list.

```
Object var = llistObj.get(2);
```

It will fetch the 3rd item from the list.

11) **Object getFirst()**: It fetches the first item from the list.

```
Object var = llistObj.getFirst();
```

12) **Object getLast()**: It fetches the last item from the list.

```
Object var= llistObj.getLast();
```

13) **int indexOf(Object item)**: It returns the index of the specified item.

```
llistObj.indexOf("bye");
```

14) **int lastIndexOf(Object item)**: It returns the index of the last occurrence of the specified element.

```
int pos = llistObj.lastIndexOf("hello);
```

integer variable pos will be having the index of last occurrence of string “hello”.

15) **Object poll()**: It returns and removes the first item of the list.

```
Object o = llistObj.poll();
```

16) **Object pollFirst()**: same as poll() method. Removes the first item of the list.

```
Object o = llistObj.pollFirst();
```

17) **Object pollLast()**: It returns and removes the last element of the list.

```
Object o = llistObj.pollLast();
```

18) **Object remove()**: It removes the first element of the list.

```
llistObj.remove();
```

19) **Object remove(int index)**: It removes the item from the list which is present at the specified index.

```
llistObj.remove(4);
```

It will remove the 5th element from the list.

20) **Object remove(Object obj)**: It removes the specified object from the list.

```
llistObj.remove("Test Item");
```

21) **Object removeFirst()**: It removes the first item from the list.

```
llistobj.removeFirst();
```

22) **Object removeLast()**: It removes the last item of the list.

```
llistObj.removeLast();
```

23) **Object removeFirstOccurrence(Object item)**: It removes the first occurrence of the specified item.

```
llistobj.removeFirstOccurrence("text");
```

It will remove the first occurrence of the string “text” from the list.

24) **Object removeLastOccurrence(Object item)**: It removes the last occurrence of the given element.

```
llistObj.removeLastOccurrence("String1);
```

It will remove the last occurrence of string “String1”.

25) **Object set(int index, Object item)**: It updates the item of specified index with the give value.

```
llistObj.set(2, "Test");
```

It will update the 3rd element with the string “Test”.

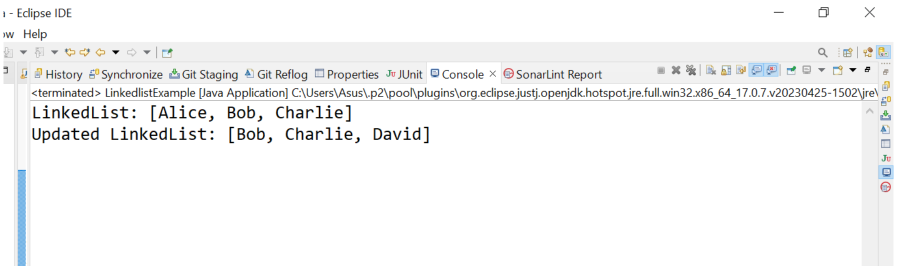
26) **int size()**: It returns the number of elements of the list.

```
llistObj.size();
```

Let us understand linked list with a programmatic example:

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

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

      linkedList.add("Alice");
      linkedList.add("Bob");
      linkedList.add("Charlie");

      System.out.println("LinkedList: " + linkedList);

      linkedList.removeFirst();
      linkedList.addLast("David");

      System.out.println("Updated LinkedList: " + linkedList);

   }
}
```

Output:



```
<terminated> LinkedlistExample [Java Application] C:\Users\Asus\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\
LinkedList: [Alice, Bob, Charlie]
Updated LinkedList: [Bob, Charlie, David]
```

Vectors :

In Java, the Vector class is a part of the Java Collections Framework and is a dynamic array-like data structure that provides synchronized access to its elements. It's similar to an ArrayList but has additional thread-safe mechanisms, making it suitable for multi-threaded environments where data consistency is a concern.

Here are key points about the Vector class:

Synchronized Operations: Vector is synchronized, which means its methods are thread-safe. This ensures that multiple threads can access and modify the vector concurrently without causing data corruption.

Dynamic Array: Like an ArrayList, a Vector also grows dynamically as elements are added to it. When the underlying array capacity is reached, it is automatically resized.

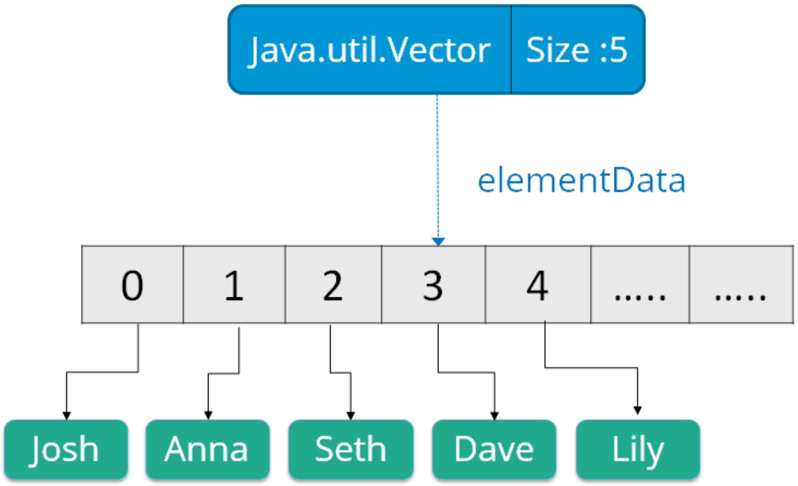
Implementation of List Interface: Vector implements the List interface, providing all the methods defined by the List interface, such as add, get, remove, and more.

Performance Considerations: Due to its synchronized nature, Vector can have performance overhead in multi-threaded scenarios compared to non-synchronized collections like ArrayList. For single-threaded applications, other collection classes might be more efficient.

Legacy Class: While Vector was widely used in earlier versions of Java, its synchronized nature led to some performance issues. With the introduction of concurrent collections, its usage has become less common.

Iteration: Iterating through a Vector using enhanced for loops or iterators is similar to iterating through other collection classes.

Use Cases: Vector might still be used in scenarios where synchronized access to data is necessary and where modern concurrent collection classes are not feasible.



Syntax:

Vector object = new Vector(size,increment);

Below are some of the methods of the Vector class:

Commonly used methods of Vector Class:

1. **void addElement(Object element):** It inserts the element at the end of the Vector.

2. **int capacity():** This method returns the current capacity of the vector.

3. **int size():** It returns the current size of the vector.

4. **void setSize(int size):** It changes the existing size with the specified size.

5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.

6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.

7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.

8. **Object firstElement():** It is used for getting the first element of the vector.

9. **Object lastElement():** Returns the last element of the array.

10. **Object get(int index):** Returns the element at the specified index.

11. **boolean isEmpty():** This method returns true if Vector doesn’t have any element.

12. **boolean removeElement(Object element):** Removes the specified element from the vector.

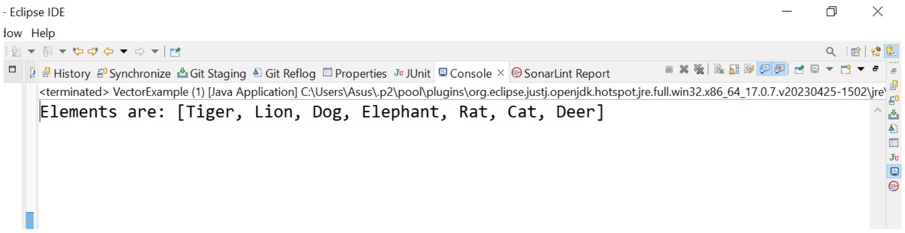
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.

14. **void setElementAt(Object element, int index):** It updates the element of specified index with the given element.

```
// Vector example

import java.util.*;

public class VectorExample {

     public static void main(String args[]) {

       //Create a vector

       Vector<String> vec = new Vector<String>();
```

```
        //Adding elements using add() method of List
        vec.add("Tiger");
        vec.add("Lion");
        vec.add("Dog");
        vec.add("Elephant");
        //Adding elements using addElement() method of Vector
        vec.addElement("Rat");
        vec.addElement("Cat");
        vec.addElement("Deer");

        System.out.println("Elements are: "+vec);
    }
}
```

Output:



```
<terminated> VectorExample (1) [Java Application] C:\Users\Asus\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
```

Knowledge check…

Now that we've covered ArrayList, LinkedList, and Vector, it's time to test your understanding. The Trainer will conduct a short poll quiz to assess your knowledge on these topics.

1. Which of the following statements is true regarding ArrayList?

a) It's a thread-safe collection.

b) It's a dynamically resizable array.

c) It's a linked list implementation.

d) It allows duplicate elements by default.

2.What type of linked list is LinkedList in Java?

a) Singly-linked list

b) Doubly-linked list

c) Circular-linked list

d) Array-based list

3. What is a key feature of the Vector class in Java?

a) It provides constant-time access by index.

b) It is a synchronized collection.

c) It uses a hash table for storage.

d) It is immutable once created.

At the end of the quiz,the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.