Collections Framework

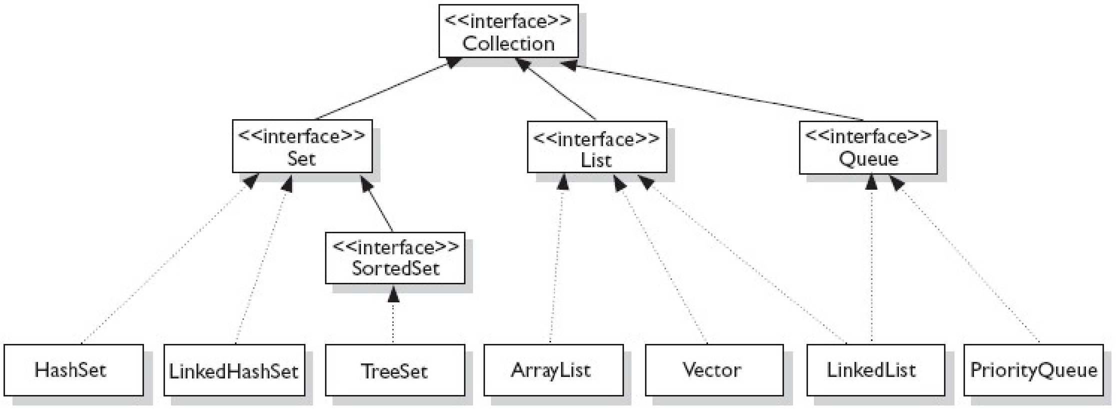
A Collection is a group of objects.

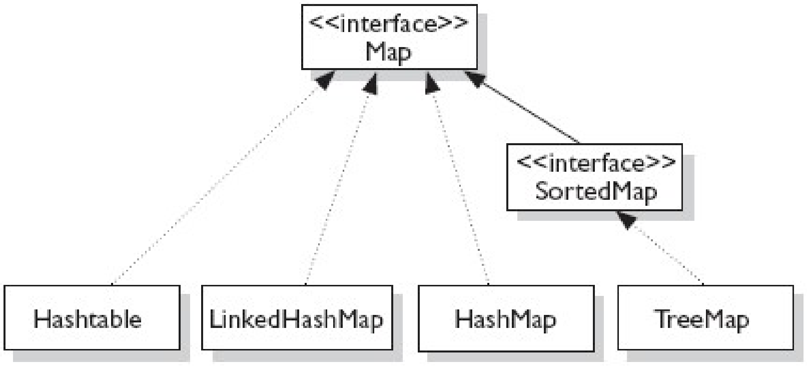
Collections framework provides a set of standard utility classes to manage collections.

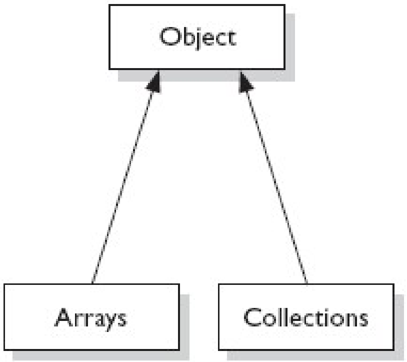
Collections Framework consists of three parts:

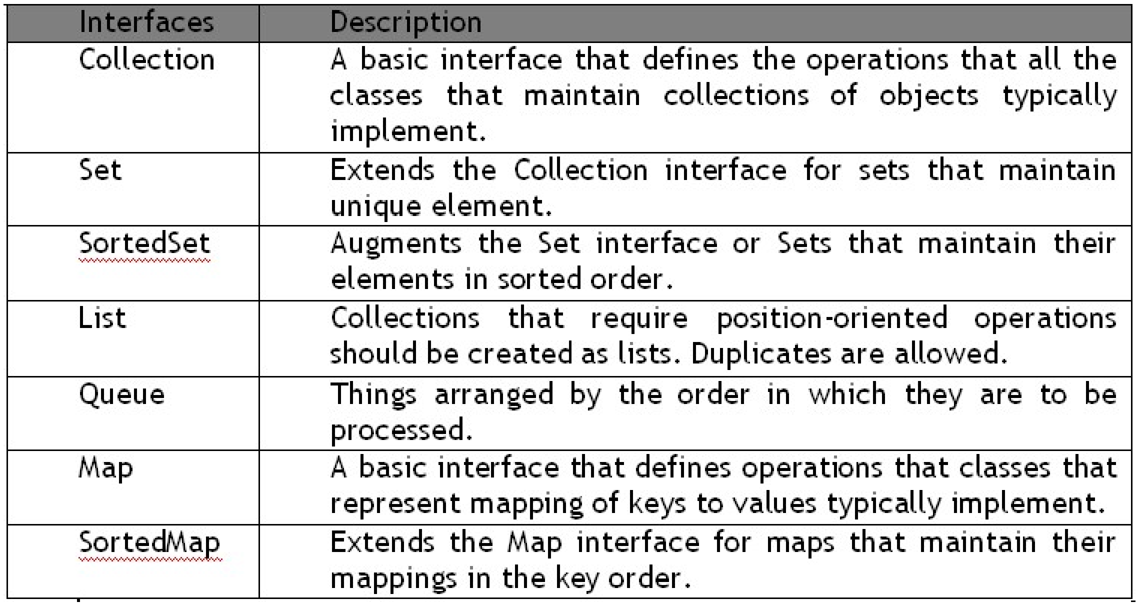
- Core Interfaces
- Concrete Implementation
- Algorithms such as searching and sorting

Collection Framework is a part of **java.util package.**









Interfaces	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends the Collection interface for sets that maintain unique element.
SortedSet	Augments the Set interface or Sets that maintain their elements in sorted order.
List	Collections that require position-oriented operations should be created as lists. Duplicates are allowed.
Queue	Things arranged by the order in which they are to be processed.
Map	A basic interface that defines operations that classes that represent mapping of keys to values typically implement.
SortedMap	Extends the Map interface for maps that maintain their mappings in the key order.

ArrayList:

ArrayList is a resizable array. It can grow and shrink dynamically as elements are added or removed.

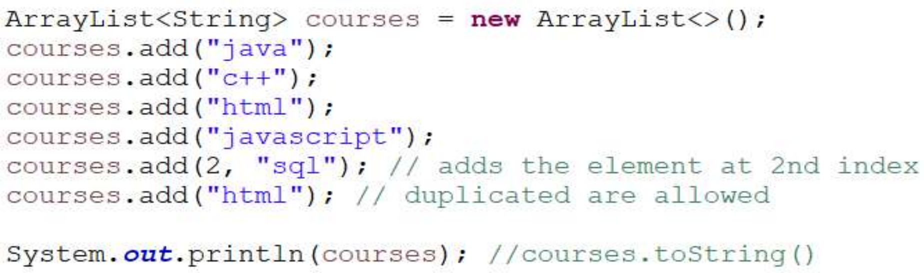
ArrayList is Indexed: It maintains the insertion order and allows random access using an index (like an array).

Some Important Methods:

1.add(E e)	Adds an element to the end of the list
2.add(int index, E element)	Inserts element at the specified index
3.get(int index)	Returns the element at the given index
4.remove(int index)	Removed the element at the specified index

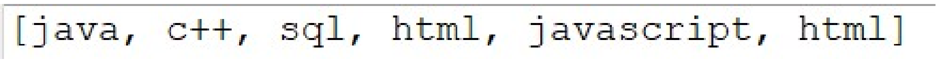
5.remove(Object o)	Removes the first occurrence of the specified object
6.size()	Returns the number of elements in the list
7.isEmpty()	Checks if the list is empty
8.contains(Object o)	Checks if the list contains the specified element
9.indexOf(Object o)	Returns the index of the first occurrence of the specified element
10.lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element

Example ArrayList of Strings:



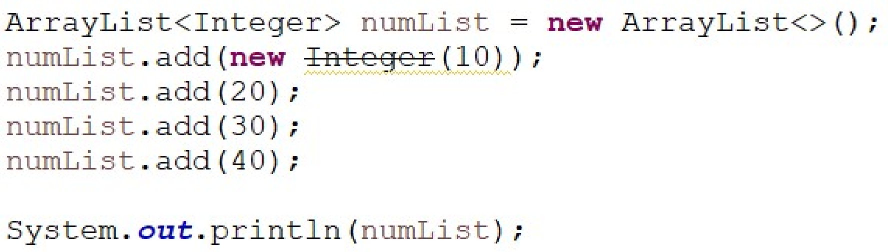
```
ArrayList<String> courses = new ArrayList<>();
courses.add("java");
courses.add("c++");
courses.add("html");
courses.add("javascript");
courses.add(2, "sql"); // adds the element at 2nd index
courses.add("html"); // duplicated are allowed

System.out.println(courses); //courses.toString()
```

OUTPUT:



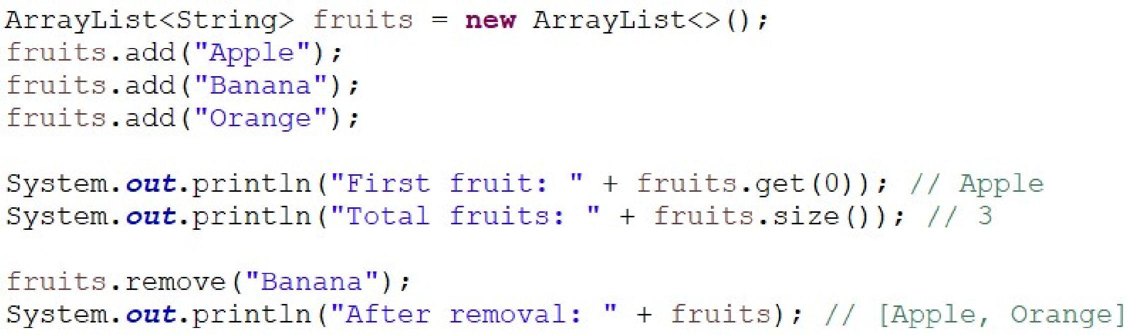
```
[java, c++, sql, html, javascript, html]
```

Example ArrayList of Integers:



```
ArrayList<Integer> numList = new ArrayList<>();
numList.add(new ~~Integer~~(10));
numList.add(20);
numList.add(30);
numList.add(40);

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

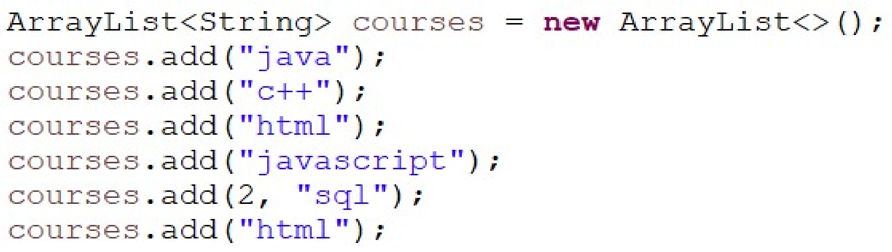
Exmaple ArrayList methods:



```
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Orange");

System.out.println("First fruit: " + fruits.get(0)); // Apple
System.out.println("Total fruits: " + fruits.size()); // 3

fruits.remove("Banana");
System.out.println("After removal: " + fruits); // [Apple, Orange]
```

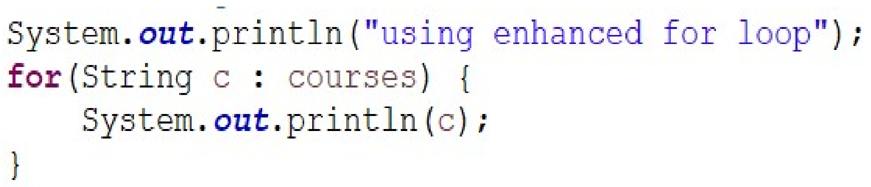
Traversing Collections:

1. Using enhanced for loop
2. Using Iterator interface
3. Using Java8 forEach loop



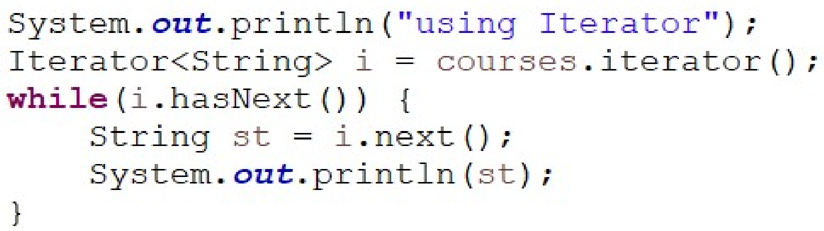
```
ArrayList<String> courses = new ArrayList<>();
courses.add("java");
courses.add("c++");
courses.add("html");
courses.add("javascript");
courses.add(2, "sql");
courses.add("html");
```

Using enhanced for loop:



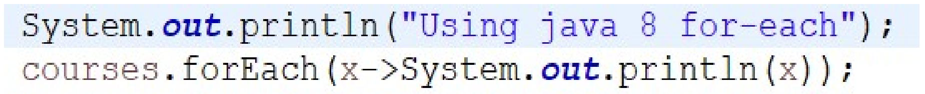
```
System.out.println("using enhanced for loop");
for(String c : courses) {
    System.out.println(c);
}
```

Using Iterator interface



```
System.out.println("using Iterator");
Iterator<String> i = courses.iterator();
while(i.hasNext()) {
    String st = i.next();
    System.out.println(st);
}
```

Using Iterator interface



```
System.out.println("Using java 8 for-each");
courses.forEach(x->System.out.println(x));
```

LinkedList:

LinkedList is a **doubly-linked list** implementation of the List interfaces in Java.

Unlike ArrayList, which uses a dynamic array internally, LinkedList stores elements as **nodes**, where each node contains:

- the data (element)
- a reference to the **next** node
- a reference to the **previous** node

Key Features of LinkedList

Faster insertions and deletions (especially in the middle or beginning)

Slower random access (you must traverse from the head or tail)

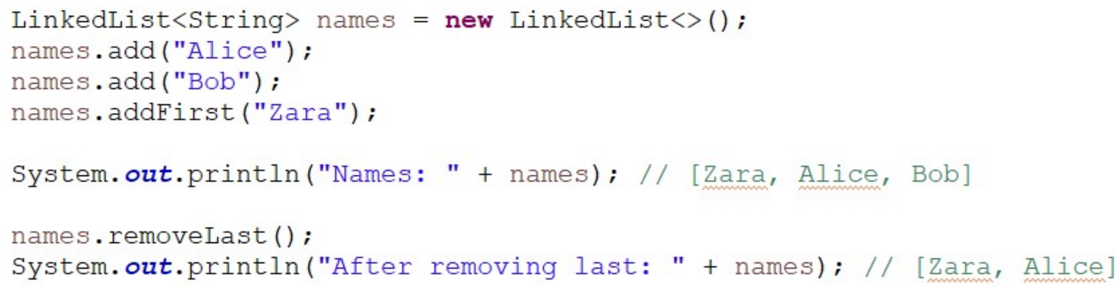
Can be used as a stack, queue, or deque

Methods in LinkedList:

Same as ArrayList, plus some queue/deque-specific methods:

- addFirst(E e) – Adds element at the beginning
- addLast(E e) – Adds element at the end
- removeFirst() / removeLast() – Removes from beginning/end
- getFirst() / getLast() – Gets first/last element

Example:



```
LinkedList<String> names = new LinkedList<>();
names.add("Alice");
names.add("Bob");
names.addFirst("Zara");

System.out.println("Names: " + names); // [Zara, Alice, Bob]

names.removeLast();
System.out.println("After removing last: " + names); // [Zara, Alice]
```

List with Custom Objects:

Product.java:

```
public class Product {

        private int productId;

        private String productName;

        private double productPrice;

        private String category;

        //constructors

        //setters and getters

}
```

Main.java:

```
public class Main {
	public static void main(String[] args) {
		Product p1 = new Product(10,” SonyHeadphone”,2000,”HeadPhone”);
		Product p2 = new Product(20,”IPhone”,80000,”mobile”);
		Product p3 = new Product(30,”DellXP”,12000,”laptop”);
		ArrayList<Product> products = new ArrayList<>();
		products.add(p1);
		products.add(p2);
		products.add(p3);
		for(Product p: products) {
			System.out.println(p.getProductId()+" "+p.getProductName()+" "+p.getProductPrice()+" "+p.getCategory());
		}
	}
}
```

Sorting Collections:

Use **Collection.sort(list)** method

This method sort the elements in ascending order by default.

Use **Collections.sort(list,Collections.reverseOrder())** to sort the Objects in Descending Order

Example 1:

```
	ArrayList<String> fruits = new ArrayList<>();
	fruits.add("Banana");
	fruits.add("Apple");
	fruits.add("Mango");
	Collections.sort(fruits); // Sort alphabetically
	System.out.println(fruits); // [Apple, Banana, Mango]
```

Exmaple 2:

```
ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(30);

numbers.add(10);

numbers.add(20);

Collections.sort(numbers); // Sort in ascending order

System.out.println(numbers); // [10, 20, 30]
```

Exmaple 3: Sort in Descending Order

```
ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(30);

numbers.add(10);

numbers.add(20);

Collections.sort(numbers, Collections.reverseOrder()); //sort in descending

System.out.println(numbers); // [30, 20, 10]
```

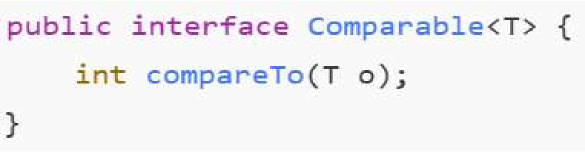
Soring Custom Objects:

Using Comparable and Comparator Interfaces we can sort the custom objects

Comparable:

Comparable is an interface **in java.lang package** that allows a class to define its **natural ordering** for sorting.

You implement this interface in your class and override the compareTo() method.



```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Returns:

- 0 → this object equals o
- Positive → this object is **greater** than o
- Negative → this object is **less** than o

Example:

Product.java

```
public class Product implements Comparable<Product> {
        private int productId;
        private String productName;
        private double productPrice;
        private String category;
        //constructors
        //setters and getters
@Override
 public int compareTo(Product p) {
     // int result = this.productId - p.getProductId(); //sort products by its id in asc order
     // int result = p.getProductId() - this.productId; //sort product by its id in desc order
     int result = this.productName.compareTo(p.getProductName()); //sort by productName
     return result;
   }
```

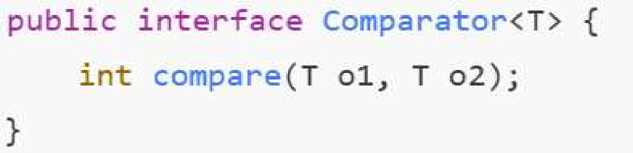
Main .java

```
     Product p1 = new Product(60,"IPhoneX",85000);
     Product p2 = new Product(30,"SamsungF20",45000);
     Product p3 = new Product(10,"SamsungM20",45000);
     Product p4 = new Product(40,"IPhone15",95000);
     ArrayList<Product> products = new ArrayList<>();
     products.add(p1);
     products.add(p2);
     products.add(p3);
     products.add(p4);
    Collections.sort(products);
     for(Product p: products) {
        System.out.println(p.getProductId()+" "+p.getProductName()+"
"+p.getProductPrice());       }
```

Comparator:

Comparator is an interface in the **java.util package** that lets you define custom sorting logic outside the class you want to sort.

It allows you to write **multiple sorting strategies** for the same object type.



```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

returns:

- 0 if o1 equals o2
- Positive if o1 > o2
- Negative if o1 < o2

To Sort the Object By using Comparator using Overloaded sort() method

Collections.sort(list, <instanceOfCompartor>);

Example:

NameCompartor.java

```
import java.util.*;

class NameComparator implements Comparator<Product> {

  @Override

  public int compare(Product p1,Product p2) {

    return p1.getProductName().compareTo(p2.getProductName());

  }

}
```

NumberComparator.java

```
import java.util.*;

class NumberComparator implements Comparator<Product> {

  @Override

  public int compare(Product p1,Product p2) {

    return p1.getProductId()-p2.getProductId();
    }
}
```

Main.java

```
        Product p1 = new Product(60,"IPhoneX",85000);
        Product p2 = new Product(30,"SamsungF20",45000);
        Product p3 = new Product(10,"SamsungM20",45000);
        Product p4 = new Product(40,"IPhone15",95000);
        ArrayList<Product> products = new ArrayList<>();
        products.add(p1);
        products.add(p2);
        products.add(p3);
        products.add(p4);
        System.out.println("Sort By Name:");
        NameComparator nameComp = new NameComparator();
        Collections.sort(products,nameComp);
        for(Product p: products) {
            System.out.println(p.getProductId()+" "+p.getProductName()+"
"+p.getProductPrice());
        }
         System.out.println("Sort By Id:");
        NumberComparator numComp = new NumberComparator();
        Collections.sort(products,numComp);
        for(Product p: products) {
            System.out.println(p.getProductId()+" "+p.getProductName()+"
"+p.getProductPrice());
        }
```