The java.util package also contains one of Java's most powerful subsystems: the Collections
Framework.
The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.

Need of collections:

int x=10; int y=20; int z=30; ....then using arrays we can declare group of values.
Student[] s=new Student[10000];
s->0.......9999
Drawback in arrays?
1 Fixed in size.
2 Arrays can hold only homogeneous data.
s[0]=new Student();
s[1] = new Customer(); wrong

But we can resolve this by using object class arrays.
Also we can create Object[] a=new Object[10000];
a[0]=new Student();
a[1]=new Customer();

3 No underlying data structure and no readymade methods available for searching and sorting
in arrays.

To overcome these problems collections are introduced.

1. Collections are growable in nature.
2. can hold both homogeneous and heterogeneous elements.
3. Every collection class is implemented based on some data structure. all readymade methods
are available.

What is a Java Collection Framework?

A Java collection framework provides an architecture to store and manipulate a group of
objects. A Java collection framework includes the following:

Interfaces
Classes
Algorithm

Interfaces: Interface in Java refers to the abstract data types. They allow Java 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.

So why do you think we need Java collections? The Java collection framework provides the
developers to access prepackaged data structures as well as algorithms to manipulate

data.

Why use Java collection?
There are several benefits of using Java collections such as:

Reducing the effort required to write the code by providing useful data structures and
algorithms

Java 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

Java Collection Framework Hierarchy - Refer to word doc

The root classes here are java.util.Collection and java.util.Map. While maps contain collection-view operations, which enable them to be manipulated as collections, from a
technical point of view, maps are not collections in Java.

java.util.Collection is not implemented directly, rather Java has implementations of its
subinterfaces.
This interface is typically used to manipulate them with max degree of generality.

Collection vs Collections in Java with Example:

java.util.Collection defines all common functionality, that sub classes should implement.

The Collection interface is not directly implemented by any class. However, it is implemented
indirectly via its subtypes or subinterfaces like List, Queue, and Set.

All the Classes of the Collection Framework implement the subInterfaces of the Collection
Interface. All the methods of Collection interfaces are also contained in it's subinterfaces.

Declaration:

public interface Collection<E> extends Iterable<E>

Type Parameters: E - the type of elements returned by this iterator

Collection vs Collections in Java with Example:

Collection is a interface present in java.util.collection. It is used to represent a group of
individual objects as a single unit. It is similar to the container in the C++ language.

The collection is considered as the root interface of the collection framework. It provides
several classes and interfaces to represent a group of individual objects as a single unit.

Collections: Collections is a utility class present in java.util.Collections. It defines several
utility methods like sorting and searching which is used to operate on collection. It has

all static methods.

For example, It has a method sort() to sort the collection elements according to default
sorting order, and it has a method min(), and max() to find the minimum and maximum value
respectively in the collection elements.

Collection vs Collections:

Collection
It is an interface.
It is used to represent a group of individual objects as a single unit.
The Collection is an interface that contains a static method since java8.
The Interface can also contain abstract and default methods.

Collections
It is a utility class.
It defines several utility methods that are used to operate on collection.
It contains only static methods.

```
// Java program to demonstrate the difference
// between Collection and Collections

import java.io.*;
import java.util.*;

class GFG {

    public static void main (String[] args)
    {

      // Creating an object of List<String>
      List<String> arrlist = new ArrayList<String>();

      // Adding elements to arrlist
      arrlist.add("geeks");
      arrlist.add("for");
      arrlist.add("geeks");

      // Printing the elements of arrlist
      // before operations
      System.out.println("Elements of arrlist before the operations:");
      System.out.println(arrlist);

      System.out.println("Elements of arrlist after the operations:");

      // Adding all the specified elements
      // to the specified collection
      Collections.addAll(arrlist, "web", "site");

      // Printing the arrlist after
      // performing addAll() method
      System.out.println(arrlist);

      // Sorting all the elements of the
      // specified collection according to
      // default sorting order
      Collections.sort(arrlist);

      // Printing the arrlist after
      // performing sort() method
      System.out.println(arrlist);

    }
}
```

Collection interface methods:
1 boolean add(Object obj)

Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not
allow duplicates.

2 boolean addAll(Collection c)

Adds all the elements of c to the invoking collection. Returns true if the operation succeeds (i.e., the elements were added). Otherwise, returns false.

3 void clear( )

Removes all elements from the invoking collection.

4 boolean contains(Object obj)

Returns true if obj is an element of the invoking collection. Otherwise, returns false.

5 boolean containsAll(Collection c)

Returns true if the invoking collection contains all elements of c. Otherwise, returns false.

6 boolean equals(Object obj)

Returns true if the invoking collection and obj are equal. Otherwise, returns false.

7 int hashCode( )

Returns the hash code for the invoking collection.

8 boolean isEmpty( )

Returns true if the invoking collection is empty. Otherwise, returns false.

9 Iterator iterator( )

Returns an iterator for the invoking collection.

10 boolean remove(Object obj)

Removes one instance of obj from the invoking collection. Returns true if the element was
removed. Otherwise, returns false.

11 boolean removeAll(Collection c)

Removes all elements of c from the invoking collection. Returns true if the collection
changed (i.e., elements were removed). Otherwise, returns false.

12 boolean retainAll(Collection c)

Removes all elements from the invoking collection except those in c. Returns true if the
collection changed (i.e., elements were removed). Otherwise, returns false.

13 int size( )

Returns the number of elements held in the invoking collection.

14 Object[ ] toArray( )

Returns an array that contains all the elements stored in the invoking collection.

The array
elements are copies of the collection elements. creates an array Element[] from the elements
of the collection

Iterator:

'Iterator' is an interface which belongs to collection framework. It allows us to traverse
the collection, access the data element and remove the data elements of the collection.
java.util package has public interface Iterator and contains three methods:

boolean hasNext(): It returns true if Iterator has more element to iterate.

Object next(): It returns the next element in the collection until the hasNext()method
return true. This method throws 'NoSuchElementException' if there is no next element.

void remove(): It removes the current element in the collection. This method throws
'IllegalStateException' if this function is called before next( ) is invoked.

```
// Java code to illustrate the use of iterator
import java.io.*;
import java.util.*;

class Test {
    public static void main(String[] args)
    {
        ArrayList<String> list = new ArrayList<String>();

        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");
        list.add("E");

        // Iterator to traverse the list
        Iterator iterator = list.iterator();

        System.out.println("List elements : ");

        while (iterator.hasNext())
            System.out.print(iterator.next() + " ");

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

Output:

```
List elements :
A B C D E
```

```
   // Iterating over collection 'c' using iterator
   for (Iterator i = c.iterator(); i.hasNext(); )
       System.out.println(i.next());
```

For eachloop is meant for traversing items in a collection.

```
   // Iterating over collection 'c' using for-each
   for (Element e: c)
       System.out.println(e);
```

We read the ':' used in for-each loop as "in". So loop reads as "for each element e in
elements", here elements is the collection which stores Element type items.

Note : In Java 8 using lambda expressions we can simply replace for-each loop with

```
elements.forEach (e -> System.out.println(e) );
```

Difference between the two traversals
In for-each loop, we can't modify collection, it will throw a ConcurrentModificationException
on the other hand with iterator we can modify collection.

Modifying a collection simply means removing an element or changing content of an item stored in the collection. This occurs because for-each loop implicitly creates an iterator but it is not exposed to the user thus we can't modify the items in the collections.

When to use which traversal?

If we have to modify collection, we can use Iterator.

While using nested for loops it is better to use for-each loop, consider the below code for
better understanding.

```
// Java program to demonstrate working of nested iterators
// may not work as expected and throw exception.
import java.util.*;

public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l = new LinkedList<Integer>();

        // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);

        // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(7);
        s.add(8);
        s.add(9);

        // Iterator to iterate over a Link List
        for (Iterator<Integer> itr1=l.iterator(); itr1.hasNext(); )
        {
            for (Iterator<Integer> itr2=s.iterator(); itr2.hasNext(); )
            {
                if (itr1.next() < itr2.next())
                {
                    System.out.println(itr1.next());
                }
            }
        }
    }
}
```

Output:

```
Exception in thread "main" java.util.NoSuchElementException

    at java.util.LinkedList$ListItr.next(LinkedList.java:888)

    at Main.main(Main.java:29)
```

The above code throws java.util.NoSuchElementException.

In the above code we are calling the next() method again and again for itr1 (i.e., for List l). Now we are advancing the iterator without even checking if it has any more
elements left in the collection(in the inner loop), thus we are advancing the iterator more
than the number of elements in the collection which leads to NoSuchElementException.

```
// Java program to demonstrate working of nested for-each
import java.util.*;
public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l=new LinkedList<Integer>();

        // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);

        // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(2);
        s.add(4);
        s.add(5);
        s.add(6);

        // Iterator to iterate over a Link List
        for (int a:l)
        {
            for (int b:s)
            {
                if (a<b)
                    System.out.print(a + " ");
            }
        }
    }
}
```

Output:

```
2 2 2 3 3 3 4 4
```

Note:
Iterator and for-each loop are faster than simple for loop for collections with no random
access, while in collections which allows random access there is no performance change with
for-each loop/for loop/iterator.

The List Interface
The java.util.List Interface extends the Collection interface.
The List interface declares methods for managing an ordered collection of object (a sequence). You can:

Control where each element is inserted in the list
Access elements by their integer index (position in the list)
Search for elements in the list
Insert duplicate elements and null values, in most List implementations

[Tip] Tip
Especially if you’re dynamically resizing a collection of object, favor using a List over a
Java array.

Some additional methods provided by List include:

add(int index, E element) — Insert an element at a specific location (without an index
argument, new elements are appended to the end)
get(int index) — Return an element at the specified location
remove(int index) — Remove an element at the specified location
set(int index, E element) — Replace the element at the specified location
subList(int fromIndex, int toIndex) — Returns as a List a modifiable view of the specified
portion of the list (that is, changes to the view actually affect the underlying list)

Classes Implementing List:
Java provides several classes that implement the List interface.
Two are used most commonly:

java.util.ArrayList
An ArrayList is usually your best bet for a List if the values remain fairly static once
you've created the list. It's more efficient than a LinkedList for random access.

java.util.LinkedList
A LinkedList provides better performance than an ArrayList if you're frequently inserting
and deleting elements, especially from the middle of the collection. But it's slower than
an ArrayList for random-access.

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 array works at the index basis.
In ArrayList, manipulation is 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.

Constructors of ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection<? extends E> c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

```
ArrayList list=new ArrayList();//creating old non-generic arraylist

ArrayList<String> list=new ArrayList<String>();//creating new generic arraylist
```

Let's see an example to traverse the ArrayList elements using the for-each loop

```
import java.util.*;
public class ArrayListExample3{
 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 for-each loop
  for(String fruit:list)
    System.out.println(fruit);

 }
```

```
}
```

The get() method returns the element at the specified index, whereas the set() method changes the element.

```
import java.util.*;
public class ArrayListExample4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Mango");
  al.add("Apple");
  al.add("Banana");
  al.add("Grapes");
  //accessing the element
  System.out.println("Returning element: "+al.get(1));
//it will return the 2nd element, because index starts from 0
  //changing the element
  al.set(1,"Dates");
  //Traversing list
  for(String fruit:al)
    System.out.println(fruit);

 }
}
```

Java ArrayList example to add elements
Here, we see different ways to add an element.

```
import java.util.*;
 class ArrayList7{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
            System.out.println("Initial list of elements: "+al);
            //Adding elements to the end of the list
            al.add("Ravi");
            al.add("Vijay");
            al.add("Ajay");
            System.out.println("After invoking add(E e) method: "+al);
            //Adding an element at the specific position
            al.add(1, "Gaurav");
            System.out.println("After invoking add(int index, E element) method:
"+al);
            ArrayList<String> al2=new ArrayList<String>();
            al2.add("Sonoo");
            al2.add("Hanumat");
            //Adding second list elements to the first list
            al.addAll(al2);
            System.out.println("After invoking addAll(Collection<? extends E> c)
method: "+al);
            ArrayList<String> al3=new ArrayList<String>();
            al3.add("John");
            al3.add("Rahul");
            //Adding second list elements to the first list at specific position
            al.addAll(1, al3);
            System.out.println("After invoking addAll(int index, Collection<? extends
E> c)
            method: "+al);

 }
}
```

ArrayList example to remove elements
Here, we see different ways to remove an element.

```
import java.util.*;
 class ArrayList8 {
```

```
        public static void main(String [] args)
        {
          ArrayList<String> al=new ArrayList<String>();
          al.add("Ravi");
          al.add("Vijay");
          al.add("Ajay");
          al.add("Anuj");
          al.add("Gaurav");
          System.out.println("An initial list of elements: "+al);
          //Removing specific element from arraylist
          al.remove("Vijay");
          System.out.println("After invoking remove(object) method: "+al);
          //Removing element on the basis of specific position
          al.remove(0);
          System.out.println("After invoking remove(index) method: "+al);
            //Creating another arraylist
          ArrayList<String> al2=new ArrayList<String>();
          al2.add("Ravi");
          al2.add("Hanumat");
          //Adding new elements to arraylist
          al.addAll(al2);
          System.out.println("Updated list : "+al);
          //Removing all the new elements from arraylist
          al.removeAll(al2);
          System.out.println("After invoking removeAll() method: "+al);
          //Removing elements on the basis of specified condition
          al.removeIf(str -> str.contains("Ajay"));   //Here, we are using Lambda
expression
          System.out.println("After invoking removeIf() method: "+al);
          //Removing all the elements available in the list
          al.clear();
          System.out.println("After invoking clear() method: "+al);
        }
    }
```

For other interfaces and its classes, refer the below links:

https://www.javatpoint.com/collections-in-java
https://www.protechtraining.com/content/java_fundamentals_tutorial-generics_collections