



CST256

Object Oriented Programming

UNIT IV



Contents

- Generics
- Generic class with two type parameter
- Bounded generics
- Collection classes: ArrayList, LinkedList, HashSet, TreeSet



Generics Introduction

- Java Generics were introduced in JDK 5.0 with the aim of reducing bugs and adding an extra layer of abstraction over types.
- The Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).
- This helps us to reuse our code.
- Generics add stability to your code by making more of your bugs detectable at compile time.”



Cont..

- In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code.
- This becomes especially apparent as new features are introduced and your code base grows in size and complexity.



Cont..

- Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there.
- Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.
- Generics add stability to your code by making more of your bugs detectable at compile time



Cont..

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements..



Generic Methods

- You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.
- we can also create a method that can be used with any type of data. Such a class is known as Generics Method.



Cont..

- Following are the rules to define Generic Methods –
- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type.
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).



Generic Class

- We can create a class that can be used with any type of data. Such a class is known as Generics Class.
- We can define our own classes with generics type. A generic type is a class or interface that is parameterized over types. We use angle brackets (< >) to specify the type parameter.

Example

```
public class GenericsType<T> {  
    private T t;  
  
    public T get(){  
        return this.t;  
    }  
  
    public void set(T t1){  
        this.t=t1;  
    }  
  
    public static void main(String args[]){  
        GenericsType<String> type = new GenericsType<>();  
        type.set("Pankaj"); //valid  
  
        GenericsType type1 = new GenericsType(); //raw  
type  
        type1.set("Pankaj"); //valid  
        type1.set(10); //valid and autoboxing support  
    }  
}
```



Generic Types

- A *generic type* is a generic class or interface that is parameterized over types.



Generics in Java

- Generics was added in Java 5 to provide **compile-time type checking** and removing risk of **ClassCastException** that was common while working with collection classes.
- The whole collection framework was re-written to **use generics for type-safety.**

```
List list = new ArrayList();  
list.add("abc");  
list.add(new Integer(5)); //OK  
  
for(Object obj : list){  
    //type casting leading to ClassCastException at runtime  
    String str=(String) obj;  
}
```



Type Safety

```
class NonGen{
    Object ob;
    NonGen(Object o)
    {      ob=o;    }
    Object ReturnObject()
    {      return ob;    }
    void ShowType()
    {      ob.getClass().getName();
    }
}
```

```
public class GenDemo {
    public static void main(String ar[])
    {
        NonGen IntOb=new NonGen(88);
        int i=(int) IntOb.ReturnObject();
        System.out.println("i=" + i);
        NonGen StrOb=new
NonGen("Hello");
        String s=(String)
StrOb.ReturnObject();
        System.out.println("s=" + s);
        IntOb=StrOb;
        i=(int) IntOb.ReturnObject();
        System.out.println("i=" + i);
    } }
```



Type Safety

```
class NonGen<T>{
    T ob;
    NonGen(T o)
    {      ob=o;    }
    T ReturnObject()
    {      return ob;    }
    void ShowType()
    {      ob.getClass().getName();
    }
}
```

```
public class GenDemo {
    public static void main(String ar[])
    {
        NonGen<Integer> IntOb=new
NonGen<>(88);
        int i= IntOb.ReturnObject();
        System.out.println("i=" + i);
        NonGen<String> StrOb=new
NonGen<>("Hello");
        String s= StrOb.ReturnObject();
        System.out.println("s=" + s);
        IntOb=StrOb;
        i= IntOb.ReturnObject();
        System.out.println("i=" + i);
    } }
```



Java Generic Interface

- Comparable interface is an example of Generics in interfaces and it's written as:

```
package java.lang;  
import java.util.*;  
  
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```



Type Parameter Naming Conventions

- Java Generic Type Naming convention helps us understanding code easily and having a **naming convention** is one of the best practices of Java programming language.
- So generics also comes with its own naming conventions. Usually, type parameter names are single, uppercase letters to make it easily distinguishable **from java variables**.
- The most commonly used type parameter names are:
 - E - Element (used extensively by the Java Collections Framework)
 - K - Key
 - N - Number
 - T - Type
 - V - Value
 - S,U,V etc. - 2nd, 3rd, 4th types



A Generic Class with Two Type Parameters

```
class TwoGen<T, V>
{
    T ob1;
    V ob2;
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2; }
    void showTypes() {
        System.out.println("Type of T is " + ob1.getClass().getName());
        System.out.println("Type of V is " +
            ob2.getClass().getName()); }
    T getob1() { return ob1; }
    V getob2() { return ob2; }
}
```



Contd..

```
class SimpGen {  
    public static void main(String args[]) {  
        TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88, "Generics")  
        // Show the types.  
        tgObj.showTypes();  
        // Obtain and show values.  
        int v = tgObj.getob1();  
        System.out.println("value: " + v);  
        String str = tgObj.getob2();  
        System.out.println("value: " + str);  
    }  
}
```



Java Generic Method

- Sometimes we don't want the whole class to be parameterized, in that case, we can create java generics method.
- Since the constructor is a special kind of method, we can use generics type in constructors too.



Example

```
public class GenericMethodTest
{
    public static < E > void printArray( E[] inputArray )
    {
        for ( E element : inputArray )
        {
            System.out.printf( "%s ", element );
        }
        System.out.println();
    }
}
```

```
public static void main( String args[] )
{
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println( "Array integerArray contains:"
);
    printArray( intArray ); // pass an Integer array

    System.out.println( "\nArray doubleArray
contains:" );
    printArray( doubleArray ); // pass a Double array

    System.out.println( "\nArray characterArray
contains:" );
    printArray( charArray ); // pass a Character array
} }
```

Cont..

```
public class GenericsMethods {  
    //Java Generic Method  
    public static <T> boolean isEqual(GenericsType<T> g1,  
GenericsType<T> g2){  
        return g1.get().equals(g2.get());  
    }  
  
    public static void main(String args[]){  
        GenericsType<String> g1 = new GenericsType<>();  
        g1.set("Pankaj");  
  
        GenericsType<String> g2 = new GenericsType<>();  
        g2.set("Pankaj");  
  
        boolean isEqual = GenericsMethods.  
<String>isEqual(g1, g2);  
        //above statement can be written simply as  
        isEqual = GenericsMethods.isEqual(g1, g2);  
        //This feature, known as type inference, allows  
you to invoke a generic method as an ordinary method, without  
specifying a type between angle brackets.  
        //Compiler will infer the type that is needed  
    }  
}
```



Java Generics Bounded Type Parameters

- type parameters can be *bounded*.
- Bounded means “*restricted*“, we can restrict types that can be accepted by a method.
- For example, we can specify that a method accepts a type and all its *subclasses (upper bound) or a type all its superclasses (lower bound)*.



Cont..


- To declare an ***upper bounded type*** we use the keyword ***extends*** after the type followed by the upper bound that we want to use.
- For example:

```
public <T extends Number> List<T> fromArrayToList(T[] a) {  
    ...  
}
```



Cont..


- There may be times when you want ***to restrict the types*** that can be used as ***type arguments in a parameterized type***.
- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what ***bounded type parameters*** are for.
- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*, which in this example is Number.
- Note: ***"extends" (as in classes) or "implements" (as in interfaces)***.



```
class Stats<T extends Number>{
    T[] nums;
    Stats(T[] o) {
        nums = o;}

    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();
        return sum / nums.length;}
}
```

```
class BoundsDemo {
    public static void main(String args[]) {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("iob average is " + v);
        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("dob average is " + w);
        // String strs[] = { "1", "2", "3", "4", "5" };
        // Stats<String> strob = new Stats<String>(strs)
        // double x = strob.average();
        // System.out.println("strob average is " + v);
    }
}
```



```
class Stats<T extends Number>{
    T[] nums;
    Stats(T[] o) {
        nums = o;}
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();
        return sum / nums.length;}
}

boolean sameAvg(Stats<T> ob) {
    if(average() == ob.average())
        return true;
    return false;
}
```

```
class BoundsDemo {
    public static void main(String args[]) {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        Stats<Double> dob = new Stats<Double>(dnums);
        if(iob.sameAvg(dob))
            System.out.println("Averages are the same.");
        else
            System.out.println("Averages differ.");
    }
}
```

Hence Use Wildcard...

```
class Stats<T extends Number>{
    T[] nums;
    Stats(T[] o) {
        nums = o;}
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();
        return sum / nums.length;}
}

boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
        return true;
    return false;
}
```

```
class BoundsDemo {
    public static void main(String args[]) {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        Stats<Double> dob = new Stats<Double>(dnums);
        if(iob.sameAvg(dob))
            System.out.println("Averages are the same.");
        else
            System.out.println("Averages differ.");
    }
}
```



Java Generics Wildcards

- *Question mark (?) is the wildcard in generics and represent an unknown type.*
- The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type.
- We can't use wildcards while invoking a generic method or instantiating a generic class.



Introduction

What are Java Collections?



A Java Collection can be defined as a unified collection of objects.

Cont..

Java Collection Framework



A Java Collection Framework is defined as a combined structure that is capable to store and apply operations to manipulate the data

A Java Collection Framework is defined as a combined structure that is capable to store and apply operations to manipulate the data

Interface

Classes

API



Collection Framework

- Collections are like **containers** that group multiple items in a single unit. For example, a jar of chocolates, list of names, etc.
- The Java **collections** framework provides a set of **interfaces and classes** to implement various data structures and algorithms.



Java Collection Interface

- The Collection interface is **the root interface of the collections framework hierarchy**.
- Java does not provide direct implementations of the Collection interface but provides implementations of its subinterfaces like **List, Set, and Queue**.



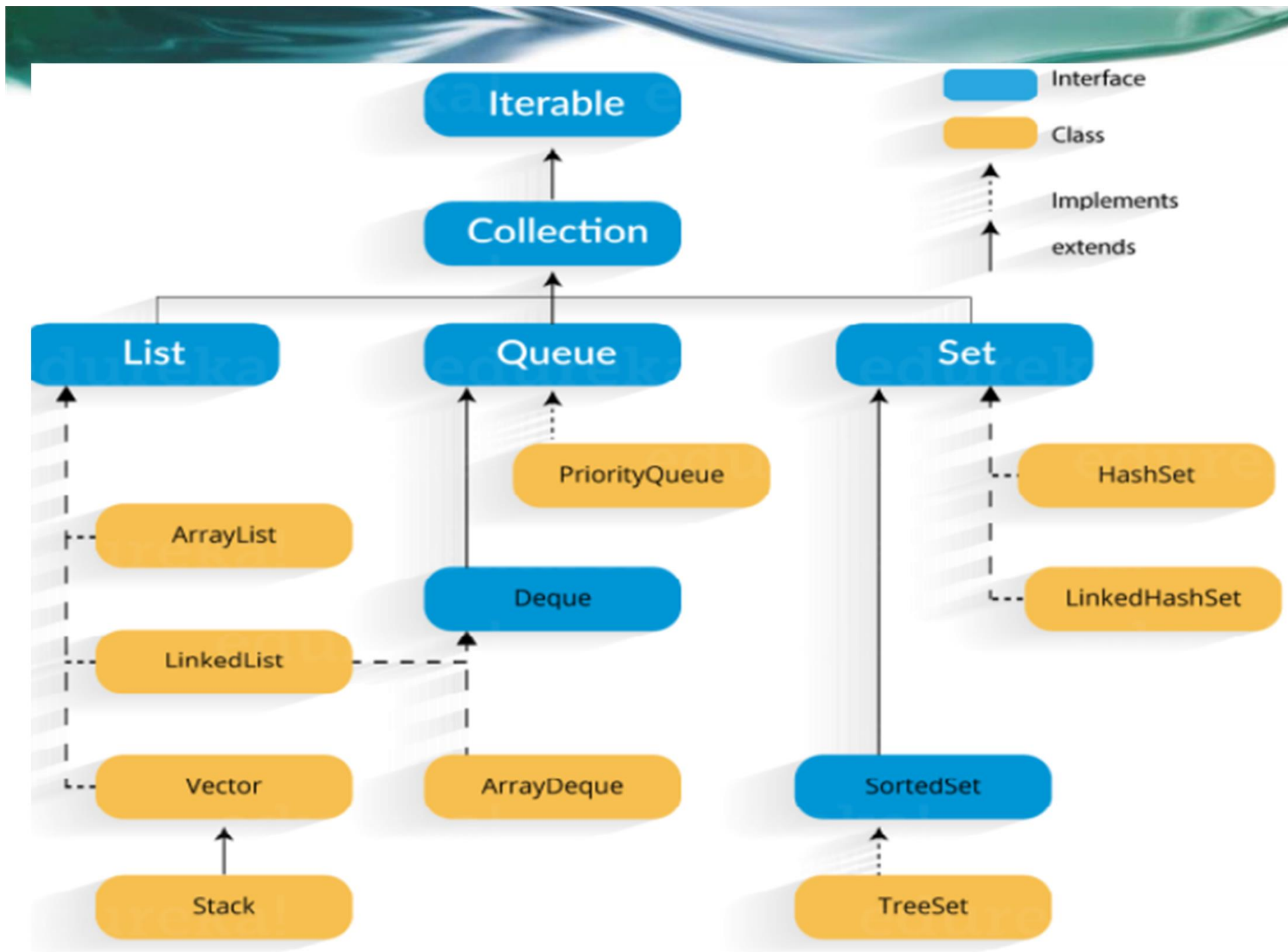
Collections Framework Vs. Collection Interface

- The Collection interface is **the root interface** of the collections framework.
- The framework includes other interfaces as well: **Map and Iterator**. These interfaces may also have **subinterfaces**.



Subinterfaces of the Collection Interface

- The Collection interface includes subinterfaces that are implemented by Java classes.
- All **the methods of the Collection interface** are also present in its subinterfaces.
- Here are the subinterfaces of the Collection Interface:
 - **List Interface**
 - **Set Interface**
 - **Queue Interface**
 - **Map Interface**
 - **Iterator Interface**





Cont...

- **Interface Collection<E>**
- **Type Parameters:E - the type of elements in this collection.**
- All Superinterfaces: [Iterable](#)<E>
- public interface **Collection**<E> extends [Iterable](#)<E>
- The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its ***elements***.
- Some collections allow **duplicate elements** and others do not. Some are **ordered and others unordered**.
- The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like Set and List.



Cont...

- **List Interface**

- The List interface is an ordered collection that allows us **to add and remove elements like an array.**

- **Set Interface**

- The Set interface allows us to store elements in different sets similar to the set in mathematics. **It cannot have duplicate elements.**

- **Queue Interface**

- The Queue interface is used when we want to store and access elements **in First In, First Out manner.**



Cont..

- **Map Interface**

- In Java, the Map interface allows elements to be **stored in key/value pairs**. Keys are unique names that can be used to access a particular element in a map. And, each key has a single value associated with it.

- **Iterator Interface**

- In Java, the Iterator interface provides methods that can be used to **access elements of collections**.



Iterator interface

- Iterator is an interface that iterates the elements. It is used to traverse the list and modify the elements.
- **public boolean hasNext()** – This method returns true if the iterator has more elements.
- **public Object next()** – It returns the element and moves the cursor pointer to the next element.
- **public void remove()** – This method removes the last elements returned by the iterator.



Java collections: List

- A List is an ordered Collection of elements which may contain **duplicates**. It is an interface that extends the Collection interface. Lists are further classified into the following:
 - ArrayList
 - LinkedList
 - Vectors



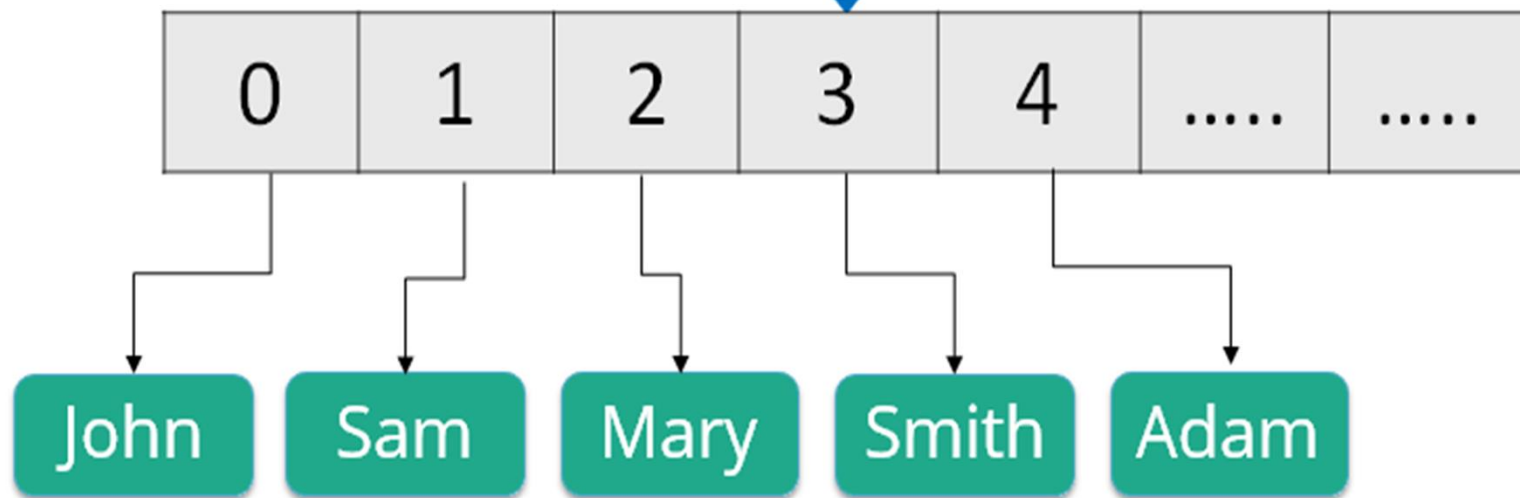
ArrayList

- **Array list:** ArrayList is the implementation of **List Interface** where the elements can be dynamically added or removed from the list.
- Also, the size of the list is increased dynamically if the elements are added more than the initial size.
- The ArrayList class allows us **to create resizable arrays.**
- **Syntax:**
- ArrayList object = new ArrayList ();



Java.util.ArrayList Size:5

elementData





Example

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

        ArrayList al=new ArrayList(); // creating array list
        al.add("Jack");                // adding elements
        al.add("Tyler");
        Iterator itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```



Example

```
ArrayList<Integer> aint=new ArrayList<>();  
    System.out.println("size of integer arraylist" + aint.size());  
    for(int i=0;i<10;i++)  
        aint.add(i+10);  
    System.out.println("size of integer arraylist" + aint.size());  
    System.out.println("arraylist " + aint);
```

Output:

```
size of integer arraylist 0  
size of integer arraylist 10  
arraylist [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

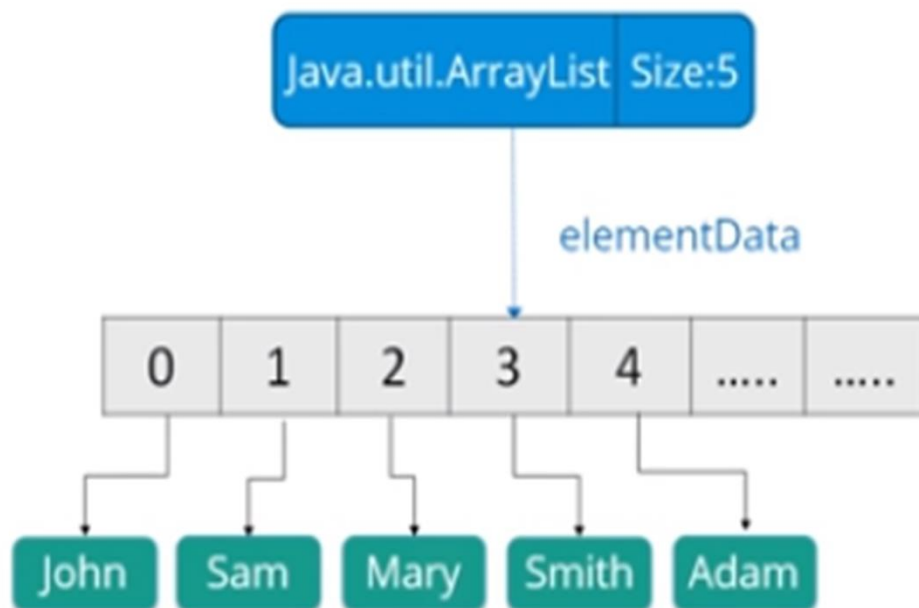
Method	Description
boolean add(Collection c)	Appends the specified element to the end of a list.
void add(int index, Object element)	Inserts the specified element at the specified position.
void clear()	Removes all the elements from this list.
int lastIndexOf(Object o)	Return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object clone()	Return a shallow copy of an ArrayList.
Object[] toArray()	Returns an array containing all the elements in the list.
void trimToSize()	Trims the capacity of this ArrayList instance to be the list's current size.

ArrayList

Hierarchy of Array List Class

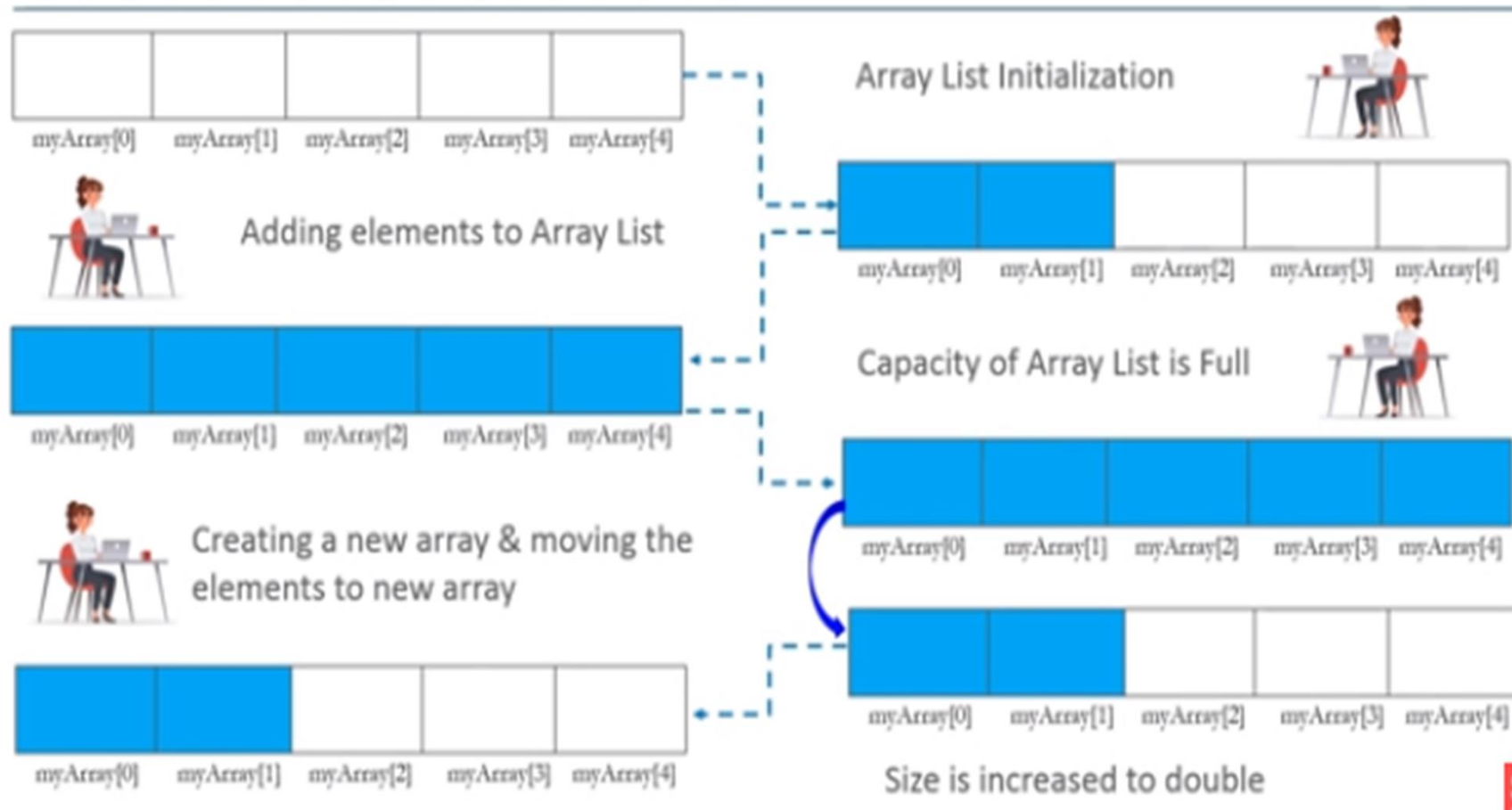


ArrayList is a part of collection framework present in `java.util` package. It provides dynamic arrays in Java.



Cont..

Internal Working of ArrayList





Cont..

Constructors of ArrayList

ArrayList()


ArrayList(Collection c)

ArrayList(int Capacity)

This constructor builds an empty array list.

Syntax:

```
ArrayList<E> myArray = new ArrayList<E>();
```



Constructor in ArrayList

ArrayList()

ArrayList(Collection c)

ArrayList(int Capacity)

This constructor builds an array list that is initialized with the elements of the collection c.

Syntax:

```
public boolean addAll(Collection c)
```

Method in ArrayList

`void add(int index, Object element)`

`void clear()`

`void trimToSize()`

`Int indexOf(Object O)`

`Object clone()`

`Object[] to Array()`

`Object remove(int index)`

This method is used to insert a specific element at a specific position index in a list.

Example:

```
ArrayList<String> al=new ArrayList<String>();  
al.add("Jino");  
al.add("Piyush");  
al.add("Pramod");
```



Cont..

void add(int index, Object element)

void clear()

void trimToSize()

Int indexOf(Object O)

Object clone()

Object[] to Array()

Object remove(int index)

Int size()

This method is used to remove all the elements from any list.

Example:

```
ArrayList<String> al=new ArrayList<String>();
```

```
al.add("abc");
```

```
al.add("xyz");
```

```
System.out.println("ArrayList before clear: "+al);
```

```
al.clear();
```

```
System.out.println("ArrayList after clear: "+al);
```



Cont..

void add(int index, Object element)

void clear()

void trimToSize()

Int indexOf(Object O)

Object clone()

Object[] to Array()

Object remove(int index)

Int size()

The **trimToSize()** method in Java trims the capacity of an ArrayList instance to be the list's current size. This method is used to trim an ArrayList instance to the number of elements it contains.

Example:

```
ArrayList<Integer> al = new ArrayList<Integer>(9);
```

```
al.add(2);
```

```
al.add(4);
```

```
al.add(5);
```

```
al.trimToSize();
```

```
System.out.println("The List elements are:");
```



Cont..

void add(int index, Object element)

void clear()

void trimToSize()

Int indexOf(Object O)

Object clone()

Object[] to Array()

Object remove(int index)

Int size()

This method returns the index of the **first occurrence** of the specified element in this list, or **-1** if this list does not contain the element.

Example:

```
ArrayList<Integer> al = new ArrayList<Integer>(9);
```

```
    al.add(2);
```

```
    al.add(4);
```

```
    al.add(5);
```

```
    int pos = al. indexOf(3);
```



Cont..

void add(int index, Object element)

void clear()

void trimToSize()

Int indexOf(Object O)

Object clone()

Object[] to Array()

Object remove(int index)

Int size()

Returns a shallow copy of this ArrayList.

Example:

```
ArrayList<String> al=new ArrayList<String>();
```

```
Object cloneList; //Added 2 elements
```

```
al.add("abc");
```

```
al.add("xyz");
```

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

```
cloneList = al.clone();
```

```
System.out.println("Elements in cloneList are:");
```




Cont..

void add(int index, Object element)

void clear()

void trimToSize()

Int indexOf(Object O)

Object clone()

Object[] to Array()

Object remove(int index)

Int size()

Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

Example:

```
arrayList.add("element_1");  
arrayList.add("element_2");  
arrayList.add("element_3");  
arrayList.add("element_4");
```

```
Object[] objArray = arrayList.toArray();
```




Cont..

<code>void add(int index, Object element)</code>	<p>java.util.ArrayList.remove(Object) method removes the first occurrence of the specified element from this list, if it is present. If the list does not contain the element, it is unchanged.</p> <p>Example:</p> <pre>arrayList.add("J"); arrayList.add("I"); arrayList.add("N"); arrayList.add("O"); arraylist.remove("N");</pre>
<code>void clear()</code>	
<code>void trimToSize()</code>	
<code>Int indexOf(Object O)</code>	
<code>Object clone()</code>	
<code>Object[] to Array()</code>	
<code>Object remove(int index)</code>	
<code>Int size()</code>	



Cont..

<code>void add(int index, Object element)</code>	<p>It returns the number of elements in this list i.e the size of the list.</p> <p>Example:</p> <p>Declaration: <code>public int size()</code></p> <pre>arrayList.add("J"); arrayList.add("I"); arrayList.add("N"); arrayList.add("O");</pre> <div><code>int asize = arraylist.size();</code></div>
<code>void clear()</code>	
<code>void trimToSize()</code>	
<code>Int indexOf(Object O)</code>	
<code>Object clone()</code>	
<code>Object[] to Array()</code>	
<code>Object remove(int index)</code>	
<code>Int size()</code>	

Cont..

Advantages of ArrayList Over Array





Cont..


Difference between ArrayList and Arrays:

ArrayList	Arrays
It is of variable-length because it is dynamic in size	It is of fixed length.
Can add different object and data into the list	Supports only primitive data-type
Allows addition of duplicate elements	Does not support a duplicate addition
Can traverse in both forward and backward direction	Can traverse only in a forward direction
Size can be modified dynamically	Size cannot be modified dynamically



Example

Write a program to create a class student with name and marks as private members. Add functionalities store and return name and marks. Create an ArrayList to store objects of student class. Create a function which takes ArrayList as input and returns the average marks of all students. Create another function which takes input as ArrayList of student and returns a new arraylist which contains the student with below average marks.



```
import java.util.ArrayList;
import java.util.Iterator;
class Student{
    private String name;
    private int marks;
    public Student(String name, int marks){
        this.name = name;
        this.marks = marks;
    }
    public String getName(){
        return this.name;
    }
    public int getMarks(){
        return this.marks;
    }
}
```

```


public class ArrayListofObject{
    public static void main(String args[]){
        //Create objects of Student class
        Student s1 = new Student("Alex", 80);
        Student s2 = new Student("Kris", 60);
        Student s3 = new Student("Peter", 90);
        //Create an ArrayList of objects Student
        ArrayList<Student> aListStudents = new
ArrayList<Student>();
        //add Objects of Student to ArrayList
        aListStudents.add(s1);
        aListStudents.add(s2);
        aListStudents.add(s3);
        //display ArrayList objects
        for(Student student : aListStudents){
            System.out.println("Name: " +
student.getName() + ", Marks:

```

```

"+student.getMarks());
        }
        double
avg=calculateaverage(aListStudents);
        System.out.println("Average of Class is " +
avg);
        ArrayList<Student>
BAVG=belowavg(aListStudents);
        System.out.println("List of Students Below
Average");
        for(Student student : BAVG){
            System.out.println("Name: " +
student.getName() + ", Marks:
"+student.getMarks());
        }
    }
}

```



```
static double  
calculateaverage(ArrayList<Student> s)  
{  
    double avg=0;  
    int count=0;  
    Iterator itr=s.iterator();  
    while(itr.hasNext())    {  
        Student s1=(Student)itr.next();  
        avg += s1.getMarks();  
        count++;  
    }  
    avg = avg/count;  
    return avg;  
}
```

```
static ArrayList<Student>  
belowavg(ArrayList<Student> s){  
    ArrayList<Student> bavg=new  
    ArrayList<Student>();  
    double avg=calculateaverage(s);  
    for(Student s1 : s ) {  
        if(s1.getMarks()<avg)  
            bavg.add(s1);  
    }  
    return bavg;  
}  
}
```




Sorting an ArrayList

```
public class integercomparable {  
    public static void main(String args[])  
    {  
        ArrayList<Integer> list = new  
ArrayList<Integer>();  
        list.add(50);    list.add(15);  
        list.add(450);    list.add(550);  
        list.add(200);  
        System.out.println(list);  
        Collections.sort(list);  
        System.out.println("ascending list: " +  
list);  
    } }  

```



Sorting an ArrayList

```
import java.util.*;
class desccomp implements
Comparator<Integer>
{
    public int compare(Integer o1, Integer
o2) {
        if(o1<o2)
            return 1;
        else
            if(o1>o2)
                return -1;
        else
            return 0;
    }
}
```

```
public class integercomparable {

    public static void main(String args[])
    {
        ArrayList<Integer> list = new
ArrayList<Integer>();
        list.add(50);    list.add(15);
        list.add(450);    list.add(550);
        list.add(200);
        System.out.println(list);
        Collections.sort(list);
        System.out.println("ascending list: " +
list);
        Collections.sort(list,new desccomp());
        System.out.println("descending list: "
+ list);
    } }
```

Sorting an object using comparable

class item implements **Comparable<item>**

```
{
    int rate;
    int quantity;
    item(int r, int q)
    {
        rate=r;
        quantity=q;
    }
    void display()
    {
        System.out.print("rate " + rate);
        System.out.println("Quantity " +
quantity);
    }
    @Override
    public int compareTo(item o) {
        if(rate > o.rate)
            return 1;
        else if (rate < o.rate)
            return -1;
        else
            return 0;
    }
}
```

```
public class arraylist {
    public static void main(String args[])
    {
        ArrayList<item> ArrItem=new
ArrayList<item>(5);
        ArrItem.add(new item(7,10));
        ArrItem.add(new item(5,10));
        ArrItem.add(new item(6,20));

        for(item i:ArrItem)
        {
            i.display();
        }
        Collections.sort(ArrItem);
        for(item i:ArrItem)
        {
            i.display();
        }
    }
}
```