# Chapter 13

# Generics and Collections

## Introduction to Generic Programming

- Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations.
- Generic programming is a style of computer programming in which algorithms are written in terms of types (data types) to-be-specified-later that are then instantiated when needed for specific types provided as parameters.
- This approach, permits writing common functions or types that differ only in the set of types on which they operate when used, which can reduce duplication in the program code.
- The distinction between generic programming and normal programming is when writing generic code the "type" (often denoted T) of the data is not explicitly stated.
- The generic programming paradigm is challenging to master because it requires a high level of abstraction (ignoring the type of data in its entirety).
- Generic programming concept is termed as parametric polymorphism in some programming languages, because it allows the language to have elements and code blocks that can have different forms based on the requirements of a program.
- Multiple languages, such as Python, Ada, C#, Delphi, and so on, employ this functionality, including Java.
- Abstract program code is perfect for use in conditions where different programming patterns need to be employed from effectively the same form of code.
- Generics are also great to ensure that programming errors can be reduced by forcing programmers to use safer methods that allow them to stay away from logical errors.
- It can be implemented as an object or method type, which is designed to cover operations on different data types, while ensuring that compiling errors are eliminated and a safety net is implemented that takes away logical errors.

# Generics Work In Java

- Java Generics are a Set of Features That Allow You to Write Code Independent of the Data Type.
- Generics are one of the important features of Java and were introduced from Java 5 onwards.
- By definition, Generics are a set of Java language features that allow the programmer to use Generic types and functions and thus ensure type safety.

# Advantages of Generics in Java

- We can write a method/class/interface once and use it for any type we want.
- We can hold only a single type of object in generics. It doesn't allow to store other objects.
- Individual Type Casting is not needed.
- By using generics, we can implement algorithms that work on different types of objects and at the same, they are type-safe too.
- It is checked at compile time so the problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

## Generic Class in Java:

- A generic class is implemented exactly like a non-generic class.
- A generic class is a class that can hold any kind of object.
- To create a Generic class we have to specify generic type <T> after the class name.
- The syntax is given below.

```
class ClassName<T> {
        // members
}
```

  - The type parameter section, delimited by angle brackets (<>), follows the class name.
  - It specifies the *type parameter* (also called *type variable*) T.
- We can specify more than one type of parameter, separated by a comma with a class.
- The syntax is given below.

```
class ClassName<T1, T2, ..., Tn> {
        // members
}
```

  - The type parameter section, delimited by angle brackets (<>), follows the class name.
  - It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.
- The classes, which accept one or more parameters, which are known as parameterized classes or parameterized types.
- Begin by examining a non-generic Box class that operates on objects of any type.
  - It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {

        private Object object;

        public void set(Object object) {

                this.object = object;
        }

        public Object get() {

                return object;
        }
}
```

- To update the Box class to use generics, you create a *generic type declaration* by changing the code "public class Box" to "public class Box<T>".
- This introduces the type variable, T, that can be used anywhere inside the class.
- With this change, the Box class becomes:

```
/* Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {

        // T stands for "Type"
        private T t;

        public void set(T t) {
                this.t = t;
        }

        public T get() {
                return t;
        }
}
```

- As you can see, all occurrences of Object are replaced by T.
- A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

## Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters.
- The most commonly used type parameter names are:
  - **E:** It denotes an element and you will find it employed by the Java Collections Framework.
  - **K:** It denotes key and will be employed for pointing to various data objects and displaying the object information accordingly.
  - **N:** It stands for number value, where we aim to employ them as parameters for the generic code.
  - **T:** It denotes type. We can employ a particular type that we want to use in the program and this will offer type safety, which is a characteristic feature of generics in Java.
  - **V:** It describes value. We can employ the type accordingly when we implement this in a generic class or object.
  - **S, U, V, etc.:** These denote that we are employing multiple types. The letters will be denoting them successively, such as S for 2nd, U for 3rd and V for 4th types.

## Invoking and Instantiating a Generic Type

- After setting up a generic class, now need to implement an invocation which needs to present a concrete value.
- To reference the generic Box class, now must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:
    - **Box<Integer> integerBox;**
  - Like any other variable declaration, this code does not actually create a new Box object.
  - Box<Integer> is read as "Box of Integer".
- It simply declares that integerBox will hold a reference to a "Box of Integer".
- To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:
    - **Box<Integer> integerBox = new Box<Integer>();**

## Generic Methods

- Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type.
- Creating a generic method is similar to creating a generic type, but the advantage here is that the scope of this type remains limited to a method.

- It is possible to set up both static and non-static generic methods, while they can also be implemented in the form of class constructions.
- If we take the example of a static generic method, we find that the type parameter must be defined before the return type of the method is mentioned in the code. For example:
    - **Pair<String, Integer>pa1 = new Pair<>("grape", 3);**

**Example** Java program to show *one type* parameters in Java Generics

```
// Java program to show working of user defined Generic classes
// We use < > to specify Parameter type
class Test<T> {
        // An object of type T is declared
        T obj;
        // constructor
        Test(T obj) {
                this.obj = obj;
        }

        public T getObject() {
                return this.obj;
        }
}

// Driver class to test above
public class MainGeneric {

        public static void main(String[] args) {
                // instance of Integer type
                Test<Integer> iObj = new Test<Integer>(15);
                System.out.println(iObj.getObject());

                // instance of String type
                Test<String> sObj = new Test<String>("Hello JAVA");
                System.out.println(sObj.getObject());
        }
}
```

**Output:**

```
15
Hello JAVA
```

**Example** Java program to show *multiple type* parameters in Java Generics

```
// Java program to show multiple type parameters in Java Generics
// We use < > to specify Parameter type
```

```
class Test<T, U> {
        T obj1;  // An object of type T
        U obj2;  // An object of type U

        // constructor
        Test(T obj1, U obj2) {
                this.obj1 = obj1;
                this.obj2 = obj2;
        }

        // To print objects of T and U
        public void print()     {
                System.out.println(obj1);
                System.out.println(obj2);
        }
}

// Driver class to test above
public class MainGeneric {

        public static void main(String[] args) {

                Test <String, Integer> obj = new Test<String, Integer>("Hello JAVA", 150);
                obj.print();
        }
}
```

**Output:**

```
Hello JAVA
150
```

## Formatting Objects for Printing with toString()

- If you print any object, java compiler internally invokes the toString() method on the object. Java "knows" that every object has a toString() method because java.lang.Object has one and all classes are ultimately subclasses of Object.

- The default implementation, in java.lang.Object, is just prints the class name, an @sign, and the object's hashCode() value.

## Example:

```
public class ToStringWithout {

        int x, y;
```

```
        /** Simple constructor */
        public ToStringWithout(int anX, int aY) {
                x = anX;
                y = aY;
        }

        /** Main just creates and prints an object */
        public static void main(String[] args) {

                ToStringWithout S=new ToStringWithout(42, 86);

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

## Output:

```
ToStringWithout@4617c264
```

# toString()

- If you want to represent any object as a string, **toString() method** comes into existence.
- The toString() method returns the string representation of the object.
- The toString() method inherited from java.lang.Object.
- So overriding the toString() method, returns the desired output.

## Example:

```
public class ToStringWith{
        int x, y;

        /** Simple constructor */
        public ToStringWith(int anX, int aY) {
                x = anX;
                y = aY;
        }

        //override the toString()
        public String toString() {
                return "ToStringWith    [" + x + ", " + y + "]";
        }
```

```
        /** Main just creates and prints an object */
        public static void main(String[] args) {

                ToStringWith S=new ToStringWith (42, 86);

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

**Output:**

```
ToStringWith    [42, 86]
```

## Overriding the equals() and hashCode() Methods

- Java.lang.object has two very important methods :
  - public boolean equals(Object obj) and
  - public int hashCode().

## equals() method

- In java equals() method is used to compare equality of two Objects.
- Simply checks if two Object references (say x and y) refer to the same Object. i.e. It checks if x == y.
- Some principles of equals() method of Object class :
  - **reflexive**
    - x.equals(x) must be true.
  - **symmetrical**
    - x.equals(y) must be true if and only if y.equals(x) is also true.
  - **transitive**
    - If x.equals(y) is true and y.equals(z) is true, then x.equals(z) must also be true.
  - **repeatable**
    - Multiple calls on x.equals(y) return the same value (unless state values used in the comparison are changed, as by calling a set method).
  - *cautious*
    - x.equals(null) must return false rather than accidentally throwing a NullPointerException.

## Example:

```
public class EqualsDemo {
        static int a = 10, b=20;
         int c;
```

```
        // Constructor
        EqualsDemo() {
                System.out.println("Addition of 10 and 20 : ");
                c = a + b;
                System.out.println("Answer : "+ c);
        }

        // Driver code
        public static void main(String args[]) {
                System.out.println("1st object created...");
                EqualsDemo obj1 = new EqualsDemo();
                System.out.println("2nd object created...");
                EqualsDemo obj2 = new EqualsDemo();
                EqualsDemo obj3 = obj1;
                System.out.println("obj1 == obj2 : " + obj1.equals(obj2));
                System.out.println("obj1 == obj3 : " + obj1.equals(obj3));
        }
}
```

## Output:

```
1st object created...
Addition of 10 and 20 :
Answer : 30
2nd object created...
Addition of 10 and 20 :
Answer : 30
obj1 == obj2 : false
obj1 == obj3 : true
```

## CompareTo method in Java

- The syntax of the **compareTo()** method is:
    - **public int compareTo(Object obj)**
- The **compareTo()** method takes a single parameter, **obj,** representing the object to be compared with the current object.
- **compareTo() Return Values**
    - For example:
        - Obj1. compareTo(Obj2)

- The Java compareTo() method compares two objects and returns an integer value with the following characteristics:
  - **Positive number:** If the Obj1 object is greater than the Obj2 object.
  - **Negative number:** If the Obj1 object is lesser than the Obj2 object.
  - **Zero:** If the strings are lexicographically equal.

- **Comparable Interface**

  - The below class implements Comparable interface and implements compareTo(Object o) method to compare two Student objects.

  - However, with this plain implementation, we have to check if the incoming object is instanceof Student and then we perform compareTo on String.

  - String implements compareTo internally so it is easier to just compare on String attributes of objects.

```
public class Student implements Comparable{
        private String name;
        @Override
        public int compareTo(Object o) {
                if(o instance of Student) {
                        Student s = (Student) o;
                        return name.compareTo(s.getName());
                }
                return 0;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
}
```

  - However, we can improve this by using the Generics version as follows:

```
public class StudentWithGenerics implements Comparable<Student>{

        private String item;
        @Override
        public int compareTo(Student s) {
                return name.compareTo(s.getName());
```

```
        }

        public String getName() {
            return item;
        }

        public void setName(String name) {
            this.name = name;
        }
    }
```

**Example:** Java Program to show how to override the compareTo() method of comparable interface

```
import java.util.*;

//implementing Comparable interface
public class Student implements Comparable<Student> {

    String name;
    int age;

    // Class constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int getage() {
        return age;
    }

    public String getname() {
        return name;
    }

    public static void main(String[] args) {

        // creating the objects
        Student obj1 = new Student("Aayush", 11);
        System.out.println("The first object Name: " + obj1.name.toString() + "
                                                    and Age: " + obj1.age);

        Student obj2 = new Student("Ravi", 12);
        System.out.println("The Second object Name: " + obj2.name.toString() + "
```

```
                                                    and Age: " + obj2.age);

                System.out.println("The details of the Student, whose age is greater");
                obj1.compareTo(obj2);

        }

        // Overriding compareTo() method
        @Override public int compareTo(Student o) {
                if (this.age > o.age) {
                        System.out.format("Name: %s \t Age: %d\n", this.name, this.age);
                        // if current object is greater,then return 1
                        return 1;
                }
                else if (this.age < o.age) {
                        System.out.format("Name: %s \t Age: %d\n", o.name, o.age);
                        // if current object is greater,then return -1
                        return -1;
                }
                else {
                        System.out.format("The objets are same");
                        // if current object is equal to o,then return 0
                        return 0;
                }
        }
}
```

**Output:**

```
The first object Name: Aayush and Age: 11
The Second object Name: Ravi and Age: 12
The details of the Student, whose age is greater
        Name: Ravi      Age: 12
```

## hashCode() method

- The hashCode method in Java is a built-in function used to return an integer hash code for the given inputs.

- There are **two** different types of Java hashCode() method which can be differentiated depending on its parameter.
  - **hashCode()** Method for objects.
  - **hashCode(int value)** Method for integers

- In Java, the hashCode() method determines the hash code for objects, pivotal for collections like HashMap and HashSet.

- The hashCode() method is supposed to return an int that should uniquely identify different objects.

- A properly written hashCode() method will follow these rules:

    - It is repeatable.
        - hashCode(x) must return the same int when called repeatedly, unless set methods have been called.

    - It is consistent with equality.
        - If x.equals(y), then x.hashCode() must == y.hashCode().

    - Distinct objects should produce distinct hashCodes
        - If !x.equals(y), it is not required that x.hashCode() != y.hashCode(), but doing so may improve performance of hash tables (i.e., hashes may call hashCode() before equals()).

- **The Java hashCode() Method**

    - The hashCode() method is defined in Java Object class which computes the hash values of given input objects.

    - It returns an integer whose value represents the hash value of the input object.

    - The hashCode() method is used to generate the hash values of objects. Using these hash values, these objects are stored in Java collections such as HashMap, HashSet and HashTable.

    - Example: this example, illustrate the basic properties of Hashing.
        - Two objects with the same value have the same hashcodes.
        - Objects with different values usually have different hashcodes.
        - Hashcodes of the same object when computed more than once must remain the same.

```
public class CheckProperties {
    public static void main(String args[]) {
        String a = "hello";
        String b = "world";

        // Printing the hashcodes of a and b
        System.out.println("HashCode of a = " + a + ": " + a.hashCode());
        System.out.println("HashCode of b = " + b + ": " + b.hashCode());
```

```
        // Declaring a different variable
        String c = "hello";

        // Printing the hashcode of c
        System.out.println("HashCode of c = " + c + ": " + c.hashCode());

        Integer a_hashcode=a.hashCode();
        Integer c_hashcode=c.hashCode();

        // Second Computation of a's hashcode
        System.out.println("HashCode of a == c: " +
                                        (a_hashcode).equals(c_hashcode));
    }
}
```

**Output:**

```
HashCode of a = hello: 99162322
HashCode of b = world: 113318802
HashCode of c = hello: 99162322
HashCode of a == c: true
```

- **The hashCode(int value) Method**
  - This method, which clearly has a parameter, determines the hashcode of its parameter: value.

```
public class IntegerHashCodeExample1 {

    public static void main(String[] args) {

        //Create integer object

        int value = 144;

        int hashValue = Integer.hashCode(value);

        System.out.println("HashCode of value of " + value + " is " + hashValue);

    }

}
```

**Example:** Java program to illustrate how hashCode() and equals() methods work. which shows how to override hashCode() method.

```
public class HashCodeTest {
```

```
public int myVar;

HashCodeTest(int myVar) {
        this.myVar = myVar;
}

public int hashCode() {
         return (myVar * 23); //here the logic is multiply by a prime number.
}

// overriding method
public boolean equals(Object o) {

        // if the object is not instanceof of HashCodeTest class,  then return false
        if(o instanceof HashCodeTest) {
                HashCodeTest hTest = (HashCodeTest) o;
                if (hTest.myVar == this.myVar) {
                        return true;
                }
                else {
                        return false;
                }
        }else {
                return false;
        }
}

public static void main(String[] args) {
        int a = 144, b= 157, c=144;
        //Create integer object
        HashCodeTest Obj1 = new HashCodeTest(a);
        HashCodeTest Obj2 = new HashCodeTest(b);
        HashCodeTest Obj3 = new HashCodeTest(c);

        System.out.println("HashCode of value of 144 is " + Obj1.hashCode());
        System.out.println("HashCode of value of 157 is " + Obj2.hashCode());
        System.out.println("HashCode of value of 144 is " + Obj3.hashCode());
        Integer Obj1_hashValue = Obj1.hashCode();
```

```
        Integer Obj2_hashValue = Obj2.hashCode();
        Integer Obj3_hashValue = Obj3.hashCode();
        System.out.println("Equality Comparision between " + a + " and " + b + "
                hash value: " + Obj1_hashValue.equals(Obj2_hashValue));
        System.out.println("Equality Comparision between " + a + " and " + c + "
                hash value: " + Obj1_hashValue.equals(Obj3_hashValue));
    }
}
```

**Output:**

```
HashCode of value of 144 is 3312
HashCode of value of 157 is 3611
HashCode of value of 144 is 3312
Equality Comparision between 144 and 157 hash value: false
Equality Comparision between 144 and 144 hash value: true
```

**Example:**

```
class SomeClass{

}

public class PrintHashCodes {

    /** Some objects to hashCode() on */
    protected static Object[] data = {
            new PrintHashCodes(),
            new java.awt.Color(0x44, 0x88, 0xcc),
            new SomeClass()
    };

    public static void main(String[] args) {

        System.out.println("About to hashCode " + data.length + " objects.");
        for (int i=0; i<data.length; i++) {
          System.out.println(data[i].toString() + " --> " + data[i].hashCode());
        }
        System.out.println("All done.");
    }
}
```

**Output:**

```
About to hashCode 3 objects.
PrintHashCodes@27d6c5e0 --> 668386784
```

```
java.awt.Color[r=68,g=136,b=204] --> -12285748
SomeClass@12edcd21 --> 317574433
All done.
```

## Collections

- Object collections are important in Java.
- The collections are available as a framework and provide the architecture required to store a collection of objects.
- The framework and the supporting APIs allow programmers to store and manipulations, such as changes and modifications.
- Figure 13.1 shows Arrays and Collections relationship with Object.

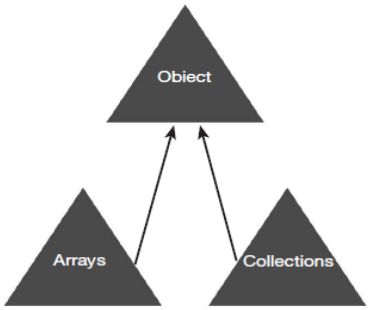



Figure 13.1 Arrays and Collections extends Object.

## Array

- Arrays hold primitive objects in Java hence it is considered as object collections.
- The capacity is assigned to an array.
- Array is managed as a singular group item.
- Thus, it is possible to sort, modify, and work on the data items that are present in an array.

## Collection

- Collections are instantiated by a programmer hence their capacity is not assigned.
- Collection hold complete objects belonging to the Primitive Wrapper classes, including Long, Integer, and Double.
- Collections do not hold the elements of the primitive types.
- The following are a few basic operations needed to perform with collections:
    - Adding new objects to the collection.
    - Removing objects from the collection.
    - Looking for objects or group of objects in the collection.
    - Getting an object from the collection without removing it.
    - Looking for a specific object at a specific index.

    - Iterating through the collection one object at a time.
- The java.util.Collection package contains all the collections implementation facilities.
- Java architecture offers three types of collections.
    - **Ordered lists:** These are important when you want to insert objects in a specific order, such as creating a waiting list.
    - **Dictionary/map:** It provides a set of information, which can be searched using a reference key to obtain the value of a particular object.
    - **Set:** It is simply unordered collection which cannot have similar objects.
- The different Ordered List are as follows:
    - **Array**
        - Array provides a storage space for their elements that have a better order.
        - Duplication is possible but each element is properly placed in its specific position.
        - All positions are available for search, and the listing can be done by either using linking or creating them in the form of an array styled listing.
    - **Stacks**
        - The stack library in Java allows the use of creating a stack of objects.
        - A stack is empty when it is created using Java implementation.
        - The stack works on the Last In, First Out (LIFO) principle, when returning objects.
        - This means that the latest item present on the stack will be the first one to be returned.
            - The top of the item will be returned first.
        - There are five operations present that ensure that any vector can be created as a stack. They are as follows:
            - The first method push()is used to place any new object in the stack.
            - The second method pop() is used to remove an object from the stack.
            - The third method is to look at the top item of the stack.
            - The fourth method finds whether a stack is empty or not.
            - The final method finds a specific item in the stack and describes its positional distance from the top of the stack.
        - **Queue**
            - This is a collection where the items are ordered.
            - All the objects are added to the end of the object list, while all removals happen from the front.
                - This presents the First In, First Out (FIFO) scheme
            - This is termed as *linked list implementation*.

## Collections Classes

- Java provides amazing collection classes that can be employed to carry out the required functions in the program.
- The lists some of the top collection classes that are available in Java.

| Map | Set | List | Queue | Utilities |
|---|---|---|---|---|
| HashMap | HashSet | ArrayList | PriorityQueue | Collections |
| Hashtable | LinkedHashSet | Vector | | Arrays |
| TreeMap | TreeSet | LinkedList | | |
| LinkedHashMap | | | | |

- The following Table shows all the properties of Collection Classes:

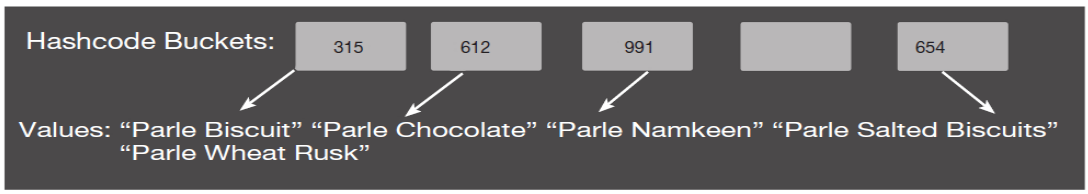
| Class | Implements | Ordered | Sorted |
|---|---|---|---|
| HashMap | Map | No | No |
| HashTable | Map | No | No |
| TreeMap | Map | Sorted | Sorted by natural order. It can also be sorted by custom comparison rules. |
| LinkedHashMap | Map | Ordered by insertion order. It can also be ordered by last access order. | No |
| HashSet | Set | No | No |
| TreeSet | Set | Sorted | Sorted by natural order. It can also be sorted by custom comparison rules. |
| LinkedHashSet | Set | Ordered by insertion order | No |
| ArrayList | List | Ordered by index | No |
| Vector | List | Ordered by index | No |
| LinkedList | List | Ordered by index | No |
| PriorityQueue | Queue | Sorted | Sorted by to-do order |

## Implementing Collection Classes

- There are several ways to implement Java Collections functionality.

## Map Interface

- Map is a key value pair collection.
- It takes unique identifier as a key that acts like an ID.
- Both key and value take object as input.
- This allows us to search for a value based on the key.
- Maps and Sets both rely on the equals() method to check if the two keys are the same or different.
- The following figure shows how values are stored as key–value pair in Map.
- Hashcode buckets contain the keys and point to its corresponding values.
- In the following example, key 315 is associated with value “Parle Biscuit”, 612 is associated with “Parle Chocolate”, etc.



- **HashMap**
  - HashMap is a collection class that uses the system of pairs, where one is the key and the other is the corresponding value.
  - The syntax is in the form of HashMap<K,V>, where the key comes first while it is followed up by the corresponding value.
  - The objects that are stored in this collection do not have to be ordered as it is employed to find any value by using the corresponding key.
  - It is possible to set up null values in this collection class.
  - This class is imported from java.util.HashMap.
  - Java HashMap class contains values based on the key.
  - Java HashMap class contains only unique keys.
  - Java HashMap class may have one null key and multiple null values.
  - Java HashMap class is non synchronized.
  - Java HashMap class maintains no order.

## Example:

```
import  java.util.*;

public class HashMap1 {
        public static void main(String[] args) {

                // creating a hash map
                HashMap<Integer,String> hm=new HashMap<Integer,String>();

                System.out.println("Initial list of elements: "+hm); //empty list

                hm.put(100,"Amit");
                hm.put(101,"Vijay");
                hm.put(102,"Rahul");

                System.out.println("After invoking put() method ");
                //print the key and its corresponding value in HashMap
                for(Map.Entry m : hm.entrySet()){
                        System.out.println(m.getKey()+" "+m.getValue());
                }

                //putIfAbsent(K,V), inserts, as the specified pair is unique
                hm.putIfAbsent(103, "Gaurav");

                System.out.println("After invoking putIfAbsent() method ");
                for(Map.Entry m : hm.entrySet()){
                        System.out.println(m.getKey()+" "+m.getValue());
                }
```

```
                //print the has map
                System.out.println("Initial list of elements: "+ hm);

                //value-based removal
                hm.remove(101);
                System.out.println("Updated list of elements: "+ hm);

                //key-value pair based removal
                hm.remove(102, "Rahul");
                System.out.println("Updated list of elements: "+ hm);
        }
}
```

## Output:

```
Initial list of elements: {} //empty list
After invoking put() method
100 Amit
101 Vijay
102 Rahul
After invoking putIfAbsent() method
100 Amit
101 Vijay
102 Rahul
103 Gaurav
Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}
Updated list of elements: {100=Amit, 102=Rahul, 103=Gaurav}
Updated list of elements: {100=Amit, 103=Gaurav}
```

## Example:

```
import java.util.HashMap
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashMapExample {
        public static void main(String[] args) {

                HashMap<Integer, String> hmap1 = new HashMap<Integer, String>();
                hmap1.put(14, "George");
                hmap1.put(33, "Paul");
                hmap1.put(16, "Jane");
                hmap1.put(7, "Brian");
                hmap1.put(19, "Jack");
```

```
        Set set1 = hmap1.entrySet();
        Iterator iterator1 = set1.iterator();
        while (iterator1.hasNext()) {
                Map.Entry ment1 = (Map.Entry) iterator1.next();
                System.out.println("The value is: "+ ment1.getValue() + " and key is: "  +
                                                                        ment1.getKey());
        }

        Object va = hmap1.get(2);
        System.out.println("Index 2 has value of " + va);

        hmap1.remove(16);

        Set set2 = hmap1.entrySet();
        Iterator iterator2 = set2.iterator();
        while (iterator2.hasNext()) {
                Map.Entry ment2 = (Map.Entry) iterator2.next();
                System.out.println("Now value is " + ment2.getValue() + " and key is: " +
                                                                        ment2.getKey());
        }
    }
}
```

## Output:

```
The value is: Jane and key is: 16
The value is: Paul and key is: 33
The value is: Jack and key is: 19
The value is: Brian and key is: 7
The value is: George and key is: 14
Index 2 has value of null
Now value is Paul and key is: 33
Now value is Jack and key is: 19
Now value is Brian and key is: 7
Now value is George and key is: 14
```

- **Hashtable**
  - Java Hashtable class implements a hashtable, which maps keys to values.
  - This class is imported from java.util.Hashtable.
  - A Hashtable is an array of a list and each list is known as a bucket.
  - The position of the bucket is identified by calling the hashcode() method.
  - A Hashtable contains values based on the key.
  - Java Hashtable class contains unique elements.
  - Java Hashtable class doesn't allow null key or value.
  - Java Hashtable class is synchronized.

**Example:**

```
import java.util.Enumeration;
import java.util.Hashtable;

public class HashtableExample {
        public static void main(String args[]) {
                Enumeration nms;
                String keys;
                Hashtable<String, String> hashtable = new Hashtable<String, String>();
                hashtable.put("Key1", "Adam");
                hashtable.put("Key2", "Brian");
                hashtable.put("Key3", "Charles");
                hashtable.put("Key4", "Dean");
                hashtable.put("Key5", "Peter");

                nms = hashtable.keys();
                while (nms.hasMoreElements()) {
                        keys = (String) nms.nextElement();
                        System.out.println("Key is " + keys + " & value is " +
                                                                        hashtable.get(keys));
                }
        }
}
```

**Output:**

```
Key is Key4 & value is Dean
Key is Key3 & value is Charles
Key is Key2 & value is Brian
Key is Key1 & value is Adam
Key is Key5 & value is Peter
```

**Example:**

```
import java.util.*;

publicclass HasTable1 {

        publicstaticvoid main(String[] args) {

                // creating a hash table
                Hashtable<Integer, String> ht=new Hashtable<Integer, String>();

                //put the key and value pair in hash table
                ht.put(100,"Amit");
                ht.put(102,"Ravi");
```

```
        ht.put(101,"Vijay");
        ht.put(103,"Rahul");
        ht.put(104,"Amit");    //Duplicate value

        //print the key and its corresponding value in hash table
        for(Map.Entry m : ht.entrySet()){
                System.out.println("Key: " + m.getKey()+ " Value: "+
                                                                m.getValue());
        }

        //print the has table
        System.out.println("Before remove: "+ ht);

        // Remove value for key 102
        ht.remove(102);
        System.out.println("After remove: "+ ht);

        //Here, we specify the if and else statement as arguments of the method
        // getOrDefault(Object key, V defaultValue)
        // It returns the value to which the specified key is mapped,
        // or defaultValue if the map contains no mapping for the key.
        System.out.println(ht.getOrDefault(101, "Not Found"));  //return Vijay
        System.out.println(ht.getOrDefault(105, "Not Found"));  //return Not Found

        //putIfAbsent(K,V), inserts, as the specified pair is unique
        ht.putIfAbsent(105,"Gaurav");
        System.out.println("Updated HashTable: "+ ht);

        //Returns the current value, as the specified pair already exist
        ht.putIfAbsent(101,"Jay");
                System.out.println("Updated Hash Table: "+ ht);
    }
}
```

**Output:**

```
Key: 104 Value: Amit
Key: 103 Value: Rahul
Key: 102 Value: Ravi
Key: 101 Value: Vijay
Key: 100 Value: Amit
Before remove: {104=Amit, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
After remove: {104=Amit, 103=Rahul, 101=Vijay, 100=Amit}
Vijay
Not Found
Updated HashTable: {105=Gaurav, 104=Amit, 103=Rahul, 101=Vijay, 100=Amit}
Updated Hash Table: {105=Gaurav, 104=Amit, 103=Rahul, 101=Vijay, 100=Amit}
```

- **TreeMap**
  - Java TreeMap contains values based on the key.
  - It implements the NavigableMap interface and extends AbstractMap class.
  - This class is imported from java.util.TreeMap.
  - Java TreeMap contains only unique elements.
  - Java TreeMap cannot have a null key but can have multiple null values.
  - Java TreeMap is non synchronized.
  - Java TreeMap maintains ascending order.

**Example:**

```
import java.util.TreeMap;

public class Main {
	public static void main(String[] args) {
		TreeMap<String, Integer> treeMap = new TreeMap<>();

			// Adding elements to the tree map
			treeMap.put("A", 1);
			treeMap.put("C", 3);
			treeMap.put("B", 2);
			System.out.println("Print the elements of TreeMap: ");
			// Iterating over the elements of the tree map
			for (String key : treeMap.keySet()) {
				System.out.println("Key: " + key + ", Value: " +
									treeMap.get(key));
			}

			System.out.println("Print the value of Key A: ");
			// Getting values from the tree map
			int valueA = treeMap.get("A");
			System.out.println("Value of A: " + valueA);

			// Removing elements from the tree map
			treeMap.remove("B");

			System.out.println("Print the elements of TreeMap after remove the
									key B: ");
			// Iterating over the elements of the tree map
			for (String key : treeMap.keySet()) {
				System.out.println("Key: " + key + ", Value: " +
									treeMap.get(key));
			}
```

```
        }
}
```

**Output:**

Print the elements of TreeMap:
Key: A, Value: 1
Key: B, Value: 2
Key: C, Value: 3
Print the value of Key A:
Value of A: 1
Print the elements of TreeMap after remove the key B:
Key: A, Value: 1
Key: C, Value: 3

**Example:**

```
import java.util.Set;
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample {
        public static void main(String[] args) {
                TreeMap<Integer, String> trmap = new TreeMap<Integer, String>();
                trmap.put(1, "Object 1");
                trmap.put(17, "Object 2");
                trmap.put(50, "Object 3");
                trmap.put(7, "Object 4");
                trmap.put(3, "Object 5");

                Set set = trmap.entrySet();
                Iterator iterator1 = set.iterator();
                while (iterator1.hasNext()) {
                        Map.Entry ment = (Map.Entry) iterator1.next();
                        System.out.println("key is: " + ment.getKey() + " and Value is: " +
                                                                        ment.getValue());
                }
        }
}
```

**Output:**

key is: 1 and Value is: Object 1
key is: 3 and Value is: Object 5
key is: 7 and Value is: Object 4

```
key is: 17 and Value is: Object 2
key is: 50 and Value is: Object 3
```

**Example:** A simple example of TreeMap

```
import java.util.Map;
import java.util.TreeMap;

public class JavaTreeMap {

	public static void main(String[] args) {

		//Creating and adding elements
		TreeMap<Integer, String> TM = new TreeMap<Integer, String>();
		TM.put(100, "Gosling");
		TM.put(101, "da Vinci");
		TM.put(102, "van Gogh");
		TM.put(103, "Java To Go");
		TM.put(104, "Vanguard");
		TM.put(105, "Darwin");
		TM.put(105, "Darwin"); // TreeSet is Set, ignores duplicates.

		System.out.printf("Our set contains %d elements", TM.size());
		System.out.println();

		// Since it is sorted we can easily get first element
		System.out.println("Lowest key is " + TM.firstKey());

		// Since it is sorted we can easily get last element
		System.out.println("Highest Key is " + TM.lastKey());

		//Traversing TreeMap in ascending
		for(Map.Entry m : TM.entrySet()){
		    System.out.println(m.getKey()+" "+m.getValue());
		}

		//display ascending set
		System.out.println("Initial Set: "+ TM);

		//display descending set
		System.out.println("Reverse Set: "+ TM.descendingKeySet());

		//Maintains descending order
		System.out.println("descendingMap: "+TM.descendingMap());
```

```
            //subMap() method return a range
            System.out.println("SubSet: "+ TM.subMap(101, true, 104, true));
        }
}
```

**Output:**

```
Our set contains 6 elements
Lowest key is 100
Highest Key is 105
100 Gosling
101 da Vinci
102 van Gogh
103 Java To Go
104 Vanguard
105 Darwin
Initial Set: {100=Gosling, 101=da Vinci, 102=van Gogh, 103=Java To Go,
104=Vanguard, 105=Darwin}
Reverse Set: [105, 104, 103, 102, 101, 100]
descendingMap: {105=Darwin, 104=Vanguard, 103=Java To Go, 102=van Gogh,
101=da Vinci, 100=Gosling}
SubSet: {101=da Vinci, 102=van Gogh, 103=Java To Go, 104=Vanguard}
```

- **LinkedHashMap**
  - LinkedHashMap class is Hashtable and Linked list implementation of the Map interface.
  - Java LinkedHashMap contains values based on the key.
  - Java LinkedHashMap contains unique elements.
  - Java LinkedHashMap may have one null key and multiple null values.
  - Java LinkedHashMap is non synchronized.
  - Java LinkedHashMap maintains insertion order.

**Example:**

```
// Importing required classes
import java.util.*;

// Main class
// IteratingOverLinkedHashMap
class LinkedHashMap1 {
    public static void main(String args[]) {

        // Initialization of a LinkedHashMap using Generics
        LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer, String>();
```

```
            // Inserting elements into Map using put() method
            lhm.put(3, "Hello...");
            lhm.put(2, "Hi...");
            lhm.put(1, "How are you?");

            // For-each loop for traversal over Map
            for (Map.Entry<Integer, String> mapElement : lhm.entrySet()) {
                        Integer key = mapElement.getKey();
                        // Finding the value using getValue() method
                        String value = mapElement.getValue();

                        // Printing the key-value pairs
                        System.out.println(key + " : " + value);
            }
      }
}
```

**Output:**

```
3 : Hello...
2 : Hi...
1 : How are you?
```

**Example:**

```
import java.util.LinkedHashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class LinkedHashMapExample {
      public static void main(String[] args) {
            // Declaring a HashMap
            LinkedHashMap<Integer, String> lihamap = new LinkedHashMap<Integer, String>();

            // Adding the elements to this collection map
            lihamap.put(21, "Abe");
            lihamap.put(35, "Drown");
            lihamap.put(1, "Jack");
            lihamap.put(3, "Karen");
            lihamap.put(100, "Lin");

            // Generating the required set
            Set set1 = lihamap.entrySet();
```

```
            // Displaying elements from this collection map
            Iterator iter1 = set1.iterator();
            while (iter1.hasNext()) {
                  Map.Entry me = (Map.Entry) iter1.next();
                  System.out.println("The key is: " + me.getKey() + " and Value is: "  +
                                                                        me.getValue());
            }
      }
}
```

**Output:**

```
The key is: 21 and Value is: Abe
The key is: 35 and Value is: Drown
The key is: 1 and Value is: Jack
The key is: 3 and Value is: Karen
The key is: 100 and Value is: Lin
```

## Set Interface

- The **set** is an interface available in the **java.util** package.
- The set interface is used to create the mathematical set.
- The **set** interface is an unordered collection of objects in which duplicate values cannot be stored.
- The **set** interface restricts the insertion of the duplicate elements.
- There are two interfaces that extend the set implementation namely
  - *SortedSet*
  - *NavigableSet*
- In order to use functionalities of the Set interface, anyone can use the following classes:
  - *HashSet*
  - *LinkedHashSet*
  - *EnumSet*
  - *TreeSet*
- Some of the commonly used methods of the Collection interface that's also available in the Set interface are:
  - **add()** - adds the specified element to the set
  - **addAll()** - adds all the elements of the specified collection to the set
  - **iterator()** - returns an iterator that can be used to access elements of the set sequentially
  - **remove()** - removes the specified element from the set
  - **removeAll()** - removes all the elements from the set that is present in another specified set

- **retainAll()** - retains all the elements in the set that are also present in another specified set
- **clear()** - removes all the elements from the set
- **size()** - returns the length (number of elements) of the set
- **toArray()** - returns an array containing all the elements of the set
- **contains()** - returns true if the set contains the specified element
- **containsAll()** - returns true if the set contains all the elements of the specified collection
- **hashCode()** - returns a hash code value (address of the element in the set)

- The Java Set interface allows us to perform basic mathematical set operations like union, intersection, and subset.
  - **Union** - to get the union of two sets *x* and *y*, we can use *x.addAll(y)*
  - **Intersection** - to get the intersection of two sets *x* and *y*, we can use *x.retainAll(y)*
  - **Subset** - to check if *x* is a subset of *y*, we can use *y.containsAll(x)*

## HashSet

- HashSet class is used to create a collection that uses a hash table for storage.
- HashSet class inherits the AbstractSet class and implements Set interface.
- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order.
  - Here, elements are inserted on the basis of their hashcode.

**Example:**

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetExample {
        public static void main(String[] args) {
                // Declaring a HashSet
                HashSet<String> haset = new HashSet<String>();

                // Adding different elements including null ones
                haset.add("Apricot");
                haset.add("Mango");
                haset.add("Orange");
                haset.add("Strawberry");
                haset.add("Dates");
```

```
                // Displaying the stored HashSet elements
                System.out.println("Accessing elements using for-each-loop: ");
                for(String x : haset){
                        System.out.println(x);
                }

                // Adding duplicate elements
                haset.add("Orange");
                haset.add("Mango");

                //remove element
                haset.remove("Dates");

                // Access Elements using iterator()
                System.out.println("Accessing elements using iterator(): ");
                Iterator<String> iterate = haset.iterator();
                while(iterate.hasNext()) {
                        System.out.println(iterate.next());
                }

                // Multiple null values
                haset.add(null);
                haset.add(null);
                // Displaying the stored HashSet elements
                System.out.println("Displaying the stored HashSet elements: ");
                System.out.println(haset);
        }
}
```

**Output:**

```
Accessing elements using for-each-loop:
Strawberry
Mango
Apricot
Dates
Orange
Accessing elements using iterator():
Strawberry
Mango
Apricot
Orange
```

Displaying the stored HashSet elements:
[null, Strawberry, Mango, Apricot, Orange]

**Example:** Java Program for set operations like, intersection, union, and difference operations.

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

        Integer[] A = {22, 45,33, 66, 55, 34, 77};
        Integer[] B = {33, 2, 83, 45, 3, 12, 55};

        Set<Integer> set1 = new HashSet<Integer>();
        set1.addAll(Arrays.asList(A));
        Set<Integer> set2 = new HashSet<Integer>();
        set2.addAll(Arrays.asList(B));

        // Finding Union of set1 and set2
        Set<Integer> union_data = new HashSet<Integer>(set1);
        union_data.addAll(set2);
        System.out.print("Union of set1 and set2 is:");
        System.out.println(union_data);

        // Finding Intersection of set1 and set2
        Set<Integer> intersection_data = new HashSet<Integer>(set1);
        intersection_data.retainAll(set2);
        System.out.print("Intersection of set1 and set2 is:");
        System.out.println(intersection_data);

        // Finding Difference of set1 and set2
        Set<Integer> difference_data = new HashSet<Integer>(set1);
        difference_data.removeAll(set2);
        System.out.print("Difference of set1 and set2 is:");
        System.out.println(difference_data);
    }
}
```

**Output:**

Union of set1 and set2 is:[33, 66, 34, 2, 83, 3, 22, 55, 12, 45, 77]
Intersection of set1 and set2 is:[33, 55, 45]
Difference of set1 and set2 is:[66, 34, 22, 77]

## LinkedHashSet

- The LinkedHashSet class of the Java collections framework provides functionalities of both the hashtable and the linked list data structure.
- It implements the Set interface.
- The LinkedHashSet maintain a doubly-linked list internally for all of its elements.
- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operations and permits null elements.
- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.
- ***Note:*** *Keeping the insertion order in the LinkedHashset has some additional costs, both in terms of extra memory and extra CPU cycles. Therefore, if it is not required to maintain the insertion order, go for the lighter-weight HashMap or the HashSet instead.*

**Example:**

```
import java.util.LinkedHashSet;

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

                // Creating an empty LinkedHashSet of string type
                LinkedHashSet<String> linkedset = new LinkedHashSet<String>();

                // add() method adding element to LinkedHashSet
                linkedset.add("D");
                linkedset.add("B");
                linkedset.add("A");
                linkedset.add("C");

                System.out.println("Original LinkedHashSet:" + linkedset);

                // size() method returns the size of LinkedHashSet using
                System.out.println("Size of LinkedHashSet = " + linkedset.size());

                // A is already exists, hence not added
                linkedset.add("A");
                linkedset.add("E");

                 // printing the updated LinkedHashMap
                System.out.println("Updated LinkedHashSet:" + linkedset);
```

```
        // Removing existing entry from above Set
        // using remove() method
        System.out.println("Removing D from LinkedHashSet: " + linkedset.remove("D"));

        // Removing entry from Set that does not exist in Set
        System.out.println( "Trying to Remove Z which is not present: " +
                                                        linkedset.remove("Z"));

        // contains() method checks the element is present inside in Set or not
        System.out.println("Checking if A is present=" + linkedset.contains("A"));

        // printing the updated LinkedHashMap
        System.out.println("Updated LinkedHashSet: " + linkedset);
    }
}
```

**Output:**

```
Original LinkedHashSet:[D, B, A, C]
Size of LinkedHashSet = 4
Updated LinkedHashSet:[D, B, A, C, E]
Removing D from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if A is present=true
Updated LinkedHashSet: [B, A, C, E]
```

## TreeSet

- Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface.
- The objects of the TreeSet class are stored in ascending order.
- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

**Example:**

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

        // Creating a String type TreeSet
```

```
        TreeSet<String> tset = new TreeSet<String>();

        // Adding various string elements to the above TreeSet
        tset.add("EFG");
        tset.add("Stores");
        tset.add("Tests");
        tset.add("Pens");
        tset.add("Ink");
        tset.add("Jane");

        // Displaying the collection of TreeSet elements
        System.out.println(tset);
    }
}
```

## Output:

[EFG, Ink, Jane, Pens, Stores, Tests]

**Example:** A simple example of TreeSet

```
import java.util.Iterator;
import java.util.TreeSet;

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

        // A TreeSet keeps objects in sorted order. Use a Comparator
        // published by String for case-insensitive sorting order.
        //Creating and adding elements
        TreeSet<String>TS = new TreeSet<>();
        TS.add("Gosling");
        TS.add("da Vinci");
        TS.add("van Gogh");
        TS.add("Java To Go");
        TS.add("Vanguard");
        TS.add("Darwin");
        TS.add("Darwin"); // TreeSet is Set, ignores duplicates.

        System.out.println("The Elements of TreeSet are: " + TS);
        System.out.printf("Our set contains %d elements", TS.size());
        System.out.println();

        // Since it is sorted we can easily get first element
        System.out.println("Lowest (alphabetically) is " + TS.first());
```

```
        // Since it is sorted we can easily get last element
        System.out.println("Highest (alphabetically) is " + TS.last());


        System.out.println("Traversing element in ascending order: ");
        Iterator<String> itr = TS.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }

        System.out.println("Traversing element in descending order: ");
        Iterator itr1=TS.descendingIterator();
        while(itr1.hasNext()){
            System.out.println(itr1.next());
        }

        //display ascending set
        System.out.println("Initial Set: "+ TS);

        //display descending set
        System.out.println("Reverse Set: "+ TS.descendingSet());

        // Print the whole list in sorted order
        System.out.println("Sorted list:");
        TS.forEach(name -> System.out.println(name));
    }
}
```

## Output:

```
The Elements of TreeSet are: [Darwin, Gosling, Java To Go, Vanguard, da Vinci, van Gogh]
Our set contains 6 elements
Lowest (alphabetically) is Darwin
Highest (alphabetically) is van Gogh
Traversing element in ascending order:
Darwin
Gosling
Java To Go
Vanguard
da Vinci
van Gogh
Traversing element in descending order:
van Gogh
```

da Vinci
Vanguard
Java To Go
Gosling
Darwin
Initial Set: [Darwin, Gosling, Java To Go, Vanguard, da Vinci, van Gogh]
Reverse Set: [van Gogh, da Vinci, Vanguard, Java To Go, Gosling, Darwin]
Sorted list:
Darwin
Gosling
Java To Go
Vanguard
da Vinci
van Gogh

## List Interface

- The List interface is found in java.util package and inherits the Collection interface.
- In Java, the List interface is an ordered collection that allows us to store and access elements sequentially.
- The List interface contains the index-based methods to insert, update, delete and search the elements.
- The List interface can have the duplicate elements also.
- The null elements can be stored in the list.
- In order to use the functionalities of the List interface, use these following classes:
    - ArrayList
    - LinkedList
    - Vector
    - Stack

## ArrayList

- Java ArrayList class uses a *dynamic array* for storing the elements.
- ArrayList is like an array, but there is *no size limit*.
- The size of ArrayList is varies according to the elements getting added or removed from the list.
- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- We can add or remove elements anytime. So, it is much more flexible than the traditional array. ArrayList is liked as an Array, but more dynamic.
- ArrayList is a standard class from java.util

- *import java.util.ArrayList;*

- To create an ArrayList
  - ArrayList al=new ArrayList();
    or
  - List al=new ArrayList();
    - Because ArrayList implements List interface

| Important methods of ArrayList | |
|---|---|
| **Method signature** | **Usage** |
| add(Object o) | Add the given element at the end |
| add(int i, Object o) | Insert the given element at the specified position |
| clear( ) | Remove all element references from the Collection |
| contains(Object o) | True if the List contains the given Object |
| get(int i) | Return the object reference at the specified position |
| indexOf(Object o) | Return the index where the given object is found, or -1 |
| remove(Object o)<br>remove(int i) | Remove an object by reference or by position |
| toArray( ) | Return an array containing the objects in the Collection |

**Example:** Demonstration of ArrayList

```
import java.util.ArrayList;

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

                // First setting the size of ArrayList
                int size = 5;

                // Now declaring ArrayList with that size
                ArrayList<Integer> arrlist = new ArrayList<Integer>(size);

                // Now Appending new elements in the list and display
                for (int i = 1; i <= size; i++) {
                        arrlist.add(i * 10);
                }
                System.out.println("Original ArrayList: " + arrlist);

                // Displaying the ArrayList after removing
                arrlist.remove(2);
                System.out.println("ArrayList after removing element at index 2: " + arrlist);
        }
}
```

**Output:**

Original ArrayList: [10, 20, 30, 40, 50]
ArrayList after removing element at index 2: [10, 20, 40, 50]

**Example:** Demonstration of ArrayList

```
import java.util.ArrayList;

publicclass ArrayList1 {
	publicstaticvoid main(String[] args) {

		//declaring ArrayList
		ArrayList arrList = newArrayList();

		// Appending the new element at the end of the list
		arrList.add(10);
		arrList.add(20);
		arrList.add(30);
		arrList.add(40);
		arrList.add(50);

		// Printing elements
		System.out.println("The initial ArrayList is: " + arrList);

		// Remove element at index 3 and print
		arrList.remove(3);
		System.out.println("After remove element, ArrayList is: " + arrList);

		// Printing elements one by one
		for (int i=0; i<arrList.size(); i++)
			System.out.print(arrList.get(i)+" ");

		System.out.println();

		//List contains the given Object
		boolean x = arrList.contains(30);
		System.out.println("List contains the given Object : " + x);

		//clear ArrayList
		arrList.clear( );
		System.out.println("After clearing element, ArrayList is: " + arrList);
	}
}
```

**Output:**

The initial ArrayList is: [10, 20, 30, 40, 50]
After remove element, ArrayList is: [10, 20, 30, 50]
10 20 30 50
List contains the given Object : true
After clearing element, ArrayList is: []

## Vector

- Vector is like the *dynamic array* which can grow or shrink its size as required.
- The Vector can store n-number of elements in it as there is no size limit.
- Like an ArrayList, it contains components that can be accessed using an integer index.
- Vector is a part of Java Collection framework and found in the java.util package.
- Vector also maintains an insertion order like an ArrayList.
- Vector is synchronized.
- Some of the Vector methods are:

| Method | Description |
|---|---|
| Add() | It is used to append the specified element in the given vector. |
| add(index, element) | adds an element to the specified position |
| addAll() | It is used to append all of the elements in the specified collection to the end of this Vector. |
| addElement() | It is used to append the specified component to the end of this vector. It increases the vector size by one. |
| get(index) | returns an element specified by the index |
| iterator() | returns an iterator object to sequentially access vector elements |
| remove(index) | removes an element from specified position. |
| removeAll() | removes all the elements |
| clear() | removes all elements. It is more efficient than removeAll() |
| contains() | searches the vector for specified element and returns a boolean result |

**Example:** Demonstration of Vector.

```
import java.util.Iterator;
import java.util.Vector;

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

```
        Vector<String> animals1= new Vector<>();

        // Using the add() method
        animals1.add("Dog");
        animals1.add("Horse");

        // Using index number
        animals1.add(2, "Cat");
        System.out.println("Vector 1: " + animals1);

        // Using addAll()
        Vector<String> animals2 = new Vector<>();
        animals2.add("Crocodile");
        animals2.add("Hourse");

        animals1.addAll(animals2);
        System.out.println("Animals Vector: " + animals1);

        // Using get()
        String element = animals1.get(2);
        System.out.println("Element at index 2: " + element);

        // Using iterator()
        Iterator<String> iterate = animals1.iterator();
        System.out.print("Elements in Vector: ");
        while(iterate.hasNext()) {
           System.out.print(iterate.next());
           System.out.print(", ");
        }
        System.out.println(" ");

        // Using remove()
        System.out.println("Removed Element: " + animals1.remove(1));
        System.out.println("Updated Animals Vector: " + animals1);

        // Using clear()
        animals1.clear();
        System.out.println("Vector after clear(): " + animals1);
    }
}
```

**Output:**

Vector 1: [Dog, Horse, Cat]
Animals Vector: [Dog, Horse, Cat, Crocodile, Hourse]
Element at index 2: Cat
Elements in Vector: Dog, Horse, Cat, Crocodile, Hourse,
Removed Element: Horse
Updated Animals Vector: [Dog, Cat, Crocodile, Hourse]
Vector after clear(): []

**Example:** Demonstration of Vector.

```
import java.util.Vector;
import java.util.Enumeration;

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

        // Setting up initial size and increments
        Vector vec = new Vector(3, 2);  // 3 is size and 2 is increament
        System.out.println("Initial size is: " + vec.size());
        System.out.println("Initial capacity is: " + vec.capacity());

        // Adding elements
        vec.addElement(new Integer(1));
        vec.addElement(new Integer(2));
        vec.addElement(new Integer(3));
        System.out.println("The vector is " + vec);
        System.out.println("Now capacity is: " + vec.capacity());

        //adding element 4 which inrease the capacity 3 to 5
        vec.addElement(new Integer(4));
        System.out.println("The vector is " + vec);
        System.out.println("Now capacity is:  " + vec.capacity());

        //adding element 6.55
        vec.addElement(new Double(6.55));
        System.out.println("The vector is " + vec);
        System.out.println("Now capacity is: " + vec.capacity());

        //adding element 9.23 and 8
        vec.addElement(new Float(9.23));
```

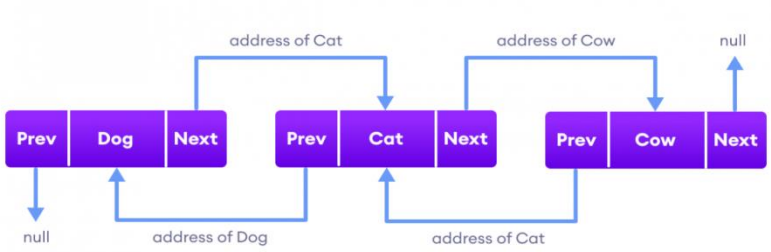
```
            vec.addElement(new Integer(8));
            System.out.println("The vector is " + vec);
            System.out.println("Now capacity is: " + vec.capacity());

            //print the first and last element
            System.out.println("First element is " + (Integer) vec.firstElement());
            System.out.println("Last element is " + (Integer) vec.lastElement());

            //check element present in the vector or not
            if (vec.contains(new Integer(3))) {
                  System.out.println("Vector contains element 3.");
            }

            // enumerate the vector elements
            Enumeration vecEnum = vec.elements();

            //Display elements
            System.out.print("Elements in the vector: ");
            while (vecEnum.hasMoreElements()){
                  System.out.print(vecEnum.nextElement() + " ");
            }
      }
}
```

**Output:**

```
Initial size is: 0
Initial capacity is: 3
The vector is [1, 2, 3]
Now capacity is: 3
The vector is [1, 2, 3, 4]
Now capacity is:  5
The vector is [1, 2, 3, 4, 6.55]
Now capacity is: 5
The vector is [1, 2, 3, 4, 6.55, 9.23, 8]
Now capacity is: 7
First element is 1
Last element is 8
Vector contains element 3.
Elements in the vector: 1 2 3 4 6.55 9.23 8
```

## LinkedList

- The LinkedList class is a collection which can contain many objects of the same type, just like the ArrayList.
- Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure.




- It inherits the AbstractList class and implements List and Deque interfaces.
- The LinkedList class has all of the same methods as the ArrayList class because they both implement the List interface.
    - This means that you can add items, change items, remove items and clear the list in the same way.
- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.
- Some of the LinkedList methods are:

| Method | Description |
|---|---|
| Add(itemt) | add item in the linked list |
| add(index, item); | add item at particular index |
| addFirst() | Adds an item to the beginning of the list |
| addLast() | Add an item to the end of the list |
| removeFirst() | Remove an item from the beginning of the list. |
| removeLast() | Remove an item from the end of the list |
| getFirst() | Get the item at the beginning of the list |
| getLast() | Get the item at the end of the list |
| set() | change item of the LinkedList. |
| contains() | checks if the LinkedList contains the item |
| indexOf() | returns the index of the first occurrence of the item |
| lastIndexOf() | returns the index of the last occurrence of the item |
| clear() | removes all the elements of the LinkedList |
| iterator() | returns an iterator to iterate over LinkedList |

**Example:** Demonstration of LinkedList.

```
import java.util.LinkedList;

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

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

                // Adding elements in various orders and positions
                obj.add("b");
                obj.add("c");
                obj.add("c");
                System.out.println("Linked list is: " + obj);

                // add the element at first and last of the List
                obj.addFirst("a");
                obj.addLast("f");
                System.out.println("Linked list is: " + obj);

                // add the element at particular index
                obj.add(4, "e");
                // change the old element by new element in list using index
                obj.set(3,"d");
                System.out.println("Linked list is: " + obj);

                // Now removing elements from the linked list using different options
                obj.remove("b");
                obj.remove(3);
                obj.removeFirst();
                obj.removeLast();
                System.out.println("New linked list after removing:  " + obj);

                // Finding elements in the linked list
                boolean stat = obj.contains("e");
                if (stat) {
                        System.out.println("List contains the element 'e' ");
                }
                else {
                        System.out.println("List doesn't contain the element 'e' ");
                }

                // find the size
                int size = obj.size();
                System.out.println("Size of linked list = " + size);
```

```
        }
}
```

**Output:**

```
Linked list is: [b, c, c]
Linked list is: [a, b, c, c, f]
Linked list is: [a, b, c, d, e, f]
New linked list after removing:  [c, d]
List doesn't contain the element 'e'
Size of linked list = 2
```

## Queue Interface

- The interface Queue is available in the java.util package and does extend the Collection interface.
- It is used to keep the elements that are processed in the First In First Out (FIFO) manner.
- It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.
- Being an interface, the queue requires, for the declaration, a concrete class, and the most common classes are the LinkedList and PriorityQueue in Java.

## PriorityQueue

- A PriorityQueue is used when the objects are supposed to be processed based on the priority.
- In the PriorityQueue the elements of the queue are needed to be processed according to the priority.
- PriorityQueue doesn’t permit null.
- We can’t create a PriorityQueue of Objects that are non-comparable
- Some of the PriorityQueue methods are:

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

**Example:** Demonstration of PriorityQueue.

```
import java.util.PriorityQueue;
import java.util.Iterator;

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

                // Creating the empty priority queue
                PriorityQueue<String> prQueue = new PriorityQueue<String>();

                // Now adding the items
                prQueue.add("C");
                prQueue.add("Java");
                prQueue.add("Python");
                prQueue.add("C++");

                // Printing priority Queue elements
                System.out.println("The elements in Priority Quee: " + prQueue);

                // Printing the most priority element
                System.out.println("The head value by using peek function is: " +
                prQueue.peek());

                //printing all elements using iterator
                System.out.println("The total queue elements are:");
                Iterator itr1 = prQueue.iterator();
                while (itr1.hasNext()){
                        System.out.println(itr1.next());
                }

                // poll() methyod remove the top priority element (or head of queue)
                prQueue.poll();
                System.out.println("After removing an element with poll function: ");
                //printing all elements using iterator
                Iterator<String> itr2 = prQueue.iterator();
                while (itr2.hasNext()) {
                        System.out.println(itr2.next());
                }

                // Removing element Java and print
                prQueue.remove("Java");
                System.out.println("after removing Java with remove function:");
                //printing all elements using iterator
                Iterator<String> itr3 = prQueue.iterator();
```

```
            while (itr3.hasNext()){
                System.out.println(itr3.next());
            }

            // Checking a particular element in the PriorityQueue
            boolean a = prQueue.contains("C");
            System.out.println("Does this priority queue contains C: " + a);
        }
}
```

**Output:**

```
The elements in Priority Quee: [C, C++, Python, Java]
The head value by using peek function is: C
The total queue elements are:
C
C++
Python
Java
After removing an element with poll function:
C++
Java
Python
after removing Java with remove function:
C++
Python
Does this priority queue contains C: false
```

## Example Array of Objects:

Write a program to create an array of String object and display it.

```
import java.util.*;

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

        //inline initialization
        String[] strArray1 = new String[] {"A","B","C"};
        String[] strArray2 = {"A","B","C"};

        //initialization after declaration
        String[] strArray3 = new String[3];
        strArray3[0] = "A";
        strArray3[1] = "B";
```

```
                strArray3[2] = "C";

                System.out.println(strArray1.equals(strArray2)); // false

                // Arrays.toString() method to convert String array to String.
                System.out.println(Arrays.toString(strArray1).equals(Arrays.toS
                                                                tring(strArray2)));// true
        }

}
```

**Output:**

false
true

**Example:** Write a program to add 3 string object to an ArrayList and display it.

```
import java.util.ArrayList;

public class ArrayList1 {
        public static void main(String[] args) {

                ArrayList aL =new ArrayList();
                aL.add("ABC");
                aL.add("PQR");
                aL.add("XYZ");

                System.out.println("The List of Names as ");
                for(int i=0; i<al.size(); i++){
                        System.out.println(aL.get(i));
                }
        }
}
```

**Output:**

The List of Names as
ABC
PQR
XYZ

**Example:** Write a program to create an ArrayList and insert 5 integer in it and find its sum.

```
import java.util.ArrayList;

publicclass ArrayList1 {
```

```
	publicstaticvoid main(String[] args) {

		//declaring ArrayList
		ArrayList arrList = new ArrayList();

		// Appending the new element at the end of the list
		arrList.add(10);
		arrList.add(20);
		arrList.add(30);
		arrList.add(40);
		arrList.add(50);

		// Printing elements
		System.out.println("The initial ArrayList is: " + arrList);

		//int sum=0;
		//for(int i=0; i<arrList.size(); i++){
		//	sum=sum + arrList.get(i);  // give compilation error why?
		//}

		int sum=0;
		for(int i=0; i<arrList.size(); i++) {
			String s = arrList.get(i).toString();
		sum=sum+Integer.parseInt(s);
		}
		System.out.println("sum="+sum);
	}
}
```

**Output:**

```
The initial ArrayList is: [10, 20, 30, 40, 50]
sum=150
```

## Sorting a Collection

- If the data into a collection in random order, and now you want to sorted.
  - If data is in an array with unsorted manner, then sort it using the static sort() method of the Arrays utility class.
    - Arrays.sort() in Java
  - If data is in a Collection, then use the static sort() method of the Collections class.

- Collections.sort() in Java

## Arrays.sort() in Java

- sort() method is a java.util.Arrays class method.

**Example:** A sample Java program to sort an array using Arrays.sort(). It by default sorts in ascending order.

```
import java.util.Arrays;

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

                //S contains 4 elements
                String[] S = {
                                "painful",
                                "mainly",
                                "gaining",
                                "raindrops"
                        };

                //sorting the array of string in ascending order
                Arrays.sort(S);
                //Display
                System.out.println("Sorted String Array: ");
                for (int i=0; i<S.length; i++) {
                        System.out.println(S[i]);
                }

                //arr contains 8 elements
                int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

                //sorting the array of integer in ascending order
                Arrays.sort(arr);
                //Convert the array of integer to String and Display
               System.out.printf("Sorted Integer Array: %s", Arrays.toString(arr));
        }
    }
```

**Output:**

```
Sorted String Array:
gaining
mainly
painful
```

raindrops
Sorted Integer Array: [6, 7, 9, 13, 21, 45, 101, 102]

**Example:** A sample Java program to sort an array in descending order

```
import java.util.Arrays;
import java.util.Collections;

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

                //S contains 4 elements
                String[] S = {
                                        "painful",
                                        "mainly",
                                        "gaining",
                                        "raindrops"
                                };

                //sorting the array of string in descending order
                Arrays.sort(S, Collections.reverseOrder());
                System.out.println("Sorted String Array: ");
                for (int i=0; i<S.length; i++) {
                        System.out.println(S[i]);
                }

                //arr contains 8 elements
                Integer[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

                //sorting the array of integer in descending order
                Arrays.sort(arr, Collections.reverseOrder());
                //Convert the array of integer to String and Display
              System.out.printf("Sorted Integer Array: %s", Arrays.toString(arr));
        }
 }
```

**Output:**

```
Sorted String Array:
raindrops
painful
mainly
gaining
Sorted Integer Array: [100, 45, 21, 13, 9, 7, 6, 2] xyz
```

## Collections.sort() in Java

- **java.util.Collections.sort()** method is present in java.util.Collections class. It is used to sort the elements present in the specified list of Collection in ascending order.

- It works similar to java.util.Arrays.sort() method but it is better then as it can sort the elements of Array as well as linked list, queue and many more present in it.

**Example:** Java program to demonstrate working of Collections.sort() to ascending and descending order.

```
import java.util.ArrayList;
import java.util.Collections;

public class CollectionSort {

    public static void main(String[] args) {

        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("yellow");
        al.add("green");
        al.add("white");
        al.add("red");
        al.add("blue");

        //sorting the elements of ArrayList in ascending order.
        Collections.sort(al);
        System.out.println("Sorted List in Ascending Order:\n" + al);

        //sorting the     elements of ArrayList in descending order.
        Collections.sort(al, Collections.reverseOrder());
        System.out.println("Sorted List in Descending Order:\n" + al);
    }
}
```

**Output:**

```
Sorted List in Ascending Order:
[blue, green, red, white, yellow]
Sorted List in Descending Order:
[yellow, white, red, green, blue]
```