**National University of Computer & Emerging Sciences, Karachi**

**Computer Science Department**

**Fall 2022, Lab Manual - 06**

|  |  |
| --- | --- |
| **Course Code: SL3001** | **Course: Software Development and construction** |
| **Instructor:** | **Miss Nida Munawar** |

**Lab # 06**

**What Are Generics?**

Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data. Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type **Integer**, **String**, **Object**

With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort.

**Generics syntax**

**class *class-name*<*type-param-list* > { // …**

3 ways for declaring a reference to a generic class and instance creation:

1. **class-name<type-arg-list > var-name = new class-name<type-arg-list >(cons-arg-list);**
2. **class-name<type-arg-list > var-name = new class-name< >(cons-arg-list);**
3. **class-name<type-arg-list > var-name = new class-name(cons-arg-list);**

**Generics Work Only with Reference Types**

When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type. You cannot use a primitive type, such as **int** or **char**. For example, with **Gen**, it is possible to pass any class type to **T**, but you cannot pass a primitive type to a type parameter. Therefore, the following declaration is illegal:

**Gen<int> intOb = new Gen<int>(53); // Error,**

can't use primitive type

Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers

## Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Sometimes you must use wrapper classes, for example when working with Generics ,Collection objects, such as ArrayList, where primitive types cannot be used (the list can only store objects):

**Java Generics Class**

**A Simple Generics Example**

**public** **class** Gen <T> {

T a;

Gen(T a1){

a= a1;

}

**void** get() {

System.***out***.println(a.getClass().getName());

}

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

Gen<Integer> b = **new** Gen<Integer>(2);

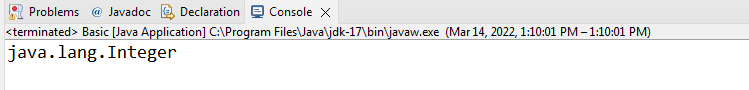
//shorten the syntax

// Gen<Integer> b = **new** Gen<>(2);

b.get();

}

}

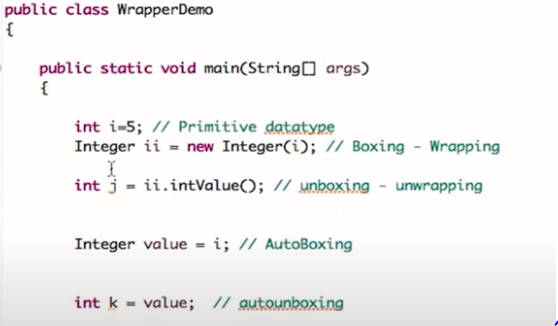


Here, **T** is the name of a *type parameter*. This name is used as a placeholder for the actual type that will be passed to **Gen** when an object is created. Thus, **T** is used within **Gen** whenever the type parameter is needed. Notice that **T** is contained within **< >**. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets. Because **Gen** uses a type parameter, **Gen** is a generic class, which is also called a *parameterized type*.

Java compiler does not actually create different versions of **Gen**, or of any other generic class. Although it’s helpful to think in these terms, it is not what actually happens. Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code *behave as if* a specific version of **Gen** were created. Thus, there is really only one version of **Gen** that actually exists in your program. The process of removing generic type information is called *erasure*

**boxing  , unboxing  , Autoboxing and Autounboxing**

**Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.**



b = new Gen<Integer>(88);

makes use of autoboxing to encapsulate the value 88, which is an **int**, into an **Integer**. This works because **Gen<Integer>** creates a constructor that takes an **Integer** argument. Because an **Integer** is expected, Java will automatically box 88 inside one. Of course, the assignment could also have been writtenexplicitly, like this:

b = new Gen<Integer>(Integer.valueOf(88));

Advantage of Java Generics

1. **Type safety**

Gen<Integer> b = **new** Gen<>(2.2);// Error!

Because **b** is of type **Gen<Integer>**, it can’t be used to refer to an object of **Gen<Double>**. This type checking is one of the main benefits of generics because it ensures type safety.

**2. Type casting is not required:**

**Generic kind of 'replaced' some casts that were needed when generic weren't there.**

**// My list of strings**

**List list = new ArrayList();**

**list.add("Hello");**

**String str1 = list.get(0); // Won't work**

**String str2 = (String) list.get(0);**

**The compiler was simply not sure that the list only contained String object, although the programmer was sure he only put Strings in it.**

**// My list of strings**

**List<String> list = new ArrayList<>(); // <> means <String> in this case**

**list.add("Hello");**

**String str1 = list.get(0); // works**

**String str2 = (String) list.get(0); // The typecast is unnecessary,**

**// for the compiler already knows**

**// that the list could only contain**

**// strings**

**So in the first code snippet, the typecast to String was necessary because you simply got an Object from the list. In the second snippet however, the typecast is made unnecessary.**

1. **Compile-Time Checking**

 It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error

**A Generic Class with Two Type Parameters**

It specifies two type parameters: **T** and **V**, separated by a comma. Because it has two type parameters, two type arguments must be passed to **TwoGen** when an object is created

**class** Gen <T , V> {

T a;

V b;

Gen(T a1 , V b1){

a= a1;

b= b1;

}

**void** get() {

System.***out***.println(a.getClass().getName());

System.***out***.println(b.getClass().getName());

}

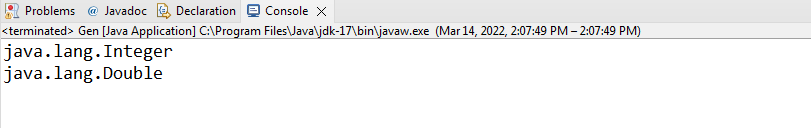
**public** **static** **void** main(String[] args) {

Gen<Integer , Double> b = **new** Gen<>(2 , 2.0);

b.get();

}

}



In this case, **Integer** is substituted for **T**, and **Double** is substituted for **V**.

Although the two type arguments differ in this example, it is possible for

both types to be the same. For example, the following line of code is valid:

Gen<Integer , Integer> b = **new** Gen<>(2 ,1);

**Bounded Types**

The type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter

Java provides *bounded types*. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an **extends** clause when specifying the type parameter, as shown here:

<*T* extends *superclass*>

This specifies that *T* can only be replaced by *superclass*, or subclasses of *superclass*. Thus, *superclass* defines an inclusive, upper limit.

**class** Gen <T **extends** Number> {

T a;

Gen(T a1 ){

a= a1;

}

**void** get() {

System.***out***.println(a.getClass().getName());

}

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

Gen<Integer> b = **new** Gen<>(2); // works fine

Gen<Double> b1 = **new** Gen<>(2.0); // works fine

Gen<Character> b2 = **new** Gen<>('a'); // error

Gen<String> b2 = **new** Gen<>("error"); // error

b.get();

}

}

**Creating a Generic Method**

As the preceding examples have shown, methods inside a generic class can make use of a class’ type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

class Main {

public static void main(String[] args) {

// initialize the class with Integer data

DemoClass demo = new DemoClass();

// generics method working with String

demo.<String>genericsMethod("Java Programming");

// generics method working with integer

demo.<Integer>genericsMethod(25);

}

}

class DemoClass {

// creae a generics method

public <T> void genericsMethod(T data) {

System.out.println("Generics Method:");

System.out.println("Data Passed: " + data);

}

}

**Note**: We can call the generics method without including the type parameter. For example,

demo.genericsMethod("Java Programming");

In this case, the compiler can match the type parameter based on the value passed to the method.

# Collections in Java

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist)

, Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist)

, [PriorityQueue](https://www.javatpoint.com/java-priorityqueue)

, HashSet, LinkedHashSet, TreeSet).

#### **What is Collection in Java**

A Collection represents a single unit of objects, i.e., a group.

#### **What is a framework in Java**

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

#### **What is Collection framework**

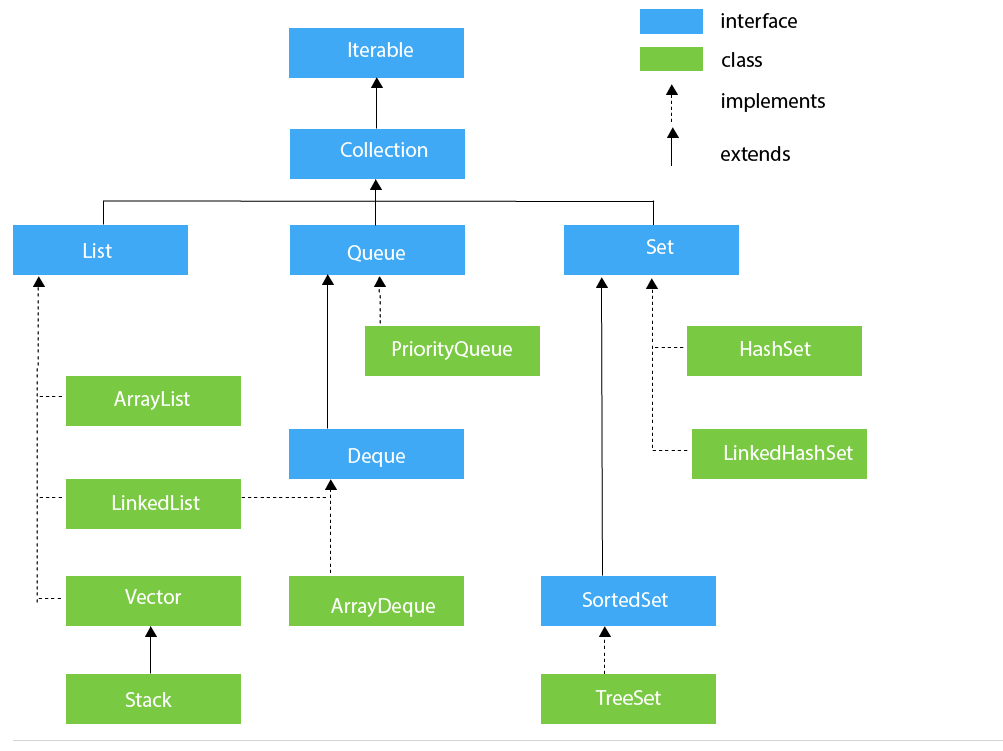
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

### Hierarchy of Collection Framework

The java.util package contains all the [classes](https://www.javatpoint.com/object-and-class-in-java)

and [interfaces](https://www.javatpoint.com/interface-in-java)



## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

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

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

## ArrayList

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

**Ex 1:**

**package** collection;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Ex\_1 {

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

List<Integer> l = **new** ArrayList();

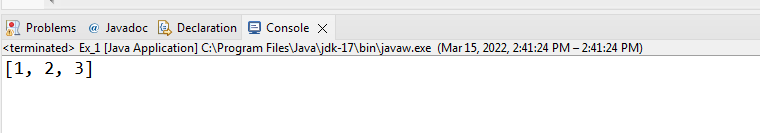
l.add(1);

l.add(2);

l.add(3);

System.***out***.println(l);

}}



**Ex 2:**

**package** collection;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Ex\_1 {

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

List<Integer> l = **new** ArrayList();

l.add(1);

l.add(2);

l.add(3);

System.***out***.println(l);

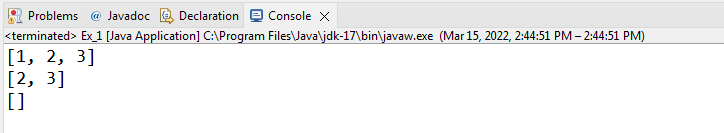
l.remove(0);

System.***out***.println(l);

l.clear();

System.***out***.println(l);

}}



**Accessing a Collection via an Iterator**

To cycle through the elements in a collection we have two ways

1. **Using an Iterator**

For example, you might want to display each element. One way to do this is to employ an *iterator*, which is an object that implements either the **Iterator** or the **ListIterator** interface. **Iterator** enables you to cycle through a collection,obtaining or removing elements. **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements. **Iterator** and **ListIterator** are generic interfaces which are declared as shown here:

interface Iterator<E>

interface ListIterator<E>

By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. **Obtain an iterator to the start of the collection by calling the collection’s iterator( ) method.**

**2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as**

**long as hasNext( ) returns true.**

1. **Within the loop, obtain each element by calling next( ).**

For collections that implement **List**, you can also obtain an iterator by

calling **listIterator( )**. As explained, a list iterator gives you the ability to

access the collection in either the forward or backward direction and lets you modify an element. Otherwise, **ListIterator** is used just like **Iterator**.

**Ex 3:**

**package** collection;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** Ex\_1 {

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

List<Integer> l = **new** ArrayList();

l.add(1);

l.add(2);

l.add(3);

Iterator i = l.iterator();

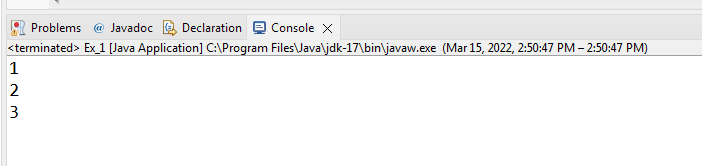
//without loop

System.***out***.println(i.next());

System.***out***.println(i.next());

System.***out***.println(i.next());

}}



**Ex 4:**

**package** collection;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** Ex\_1 {

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

List<Integer> l = **new** ArrayList();

l.add(1);

l.add(2);

l.add(3);

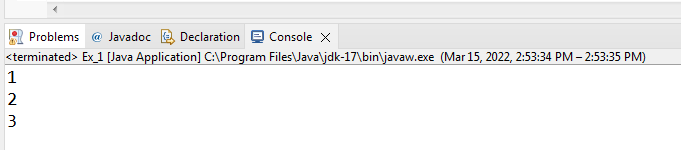
Iterator<Integer> i = l.iterator();//you can also specify type

//with loop

**while**(i.hasNext())

System.***out***.println(i.next());

}}



**ListIterator for modification**

**package** collection;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.ListIterator;

**public** **class** Ex\_1 {

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

List<String> l = **new** ArrayList();

l.add("A");

l.add("B");

//modify objects using listiterator

ListIterator<String> i = l.listIterator();

**while**(i.hasNext()) {

String e = i.next();

i.set(e + "+");

}

//printing

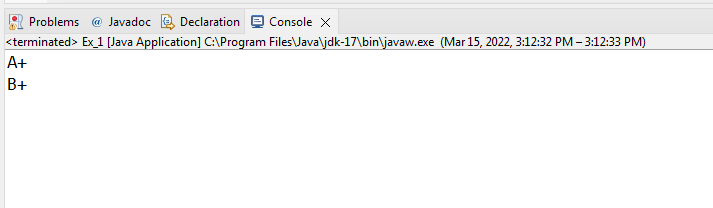
i = l.listIterator();

**while**(i.hasNext()) {

System.***out***.println(i.next());

}

}}



**ListIterator for Backward display**

**package** collection;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.ListIterator;

**public** **class** Ex\_1 {

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

List<String> l = **new** ArrayList();

l.add("A");

l.add("B");

//modify objects using listiterator

ListIterator<String> i = l.listIterator();

**while**(i.hasNext()) {

String e = i.next();

i.set(e + "+");

}

//printing

i = l.listIterator();

**while**(i.hasNext()) {

System.***out***.println(i.next());

}

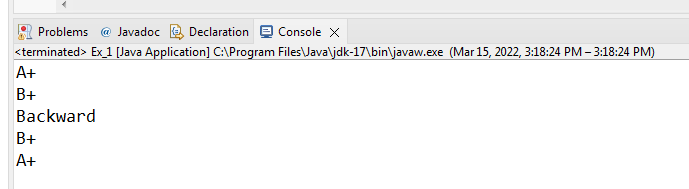
System.***out***.println("Backward");

**while**(i.hasPrevious()) {

System.***out***.println(i.previous());

}

}}



**2.The For-Each Alternative to Iterators**

If you won’t be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the **for** loop is often a more convenient alternative to cycling through a collection than is using an iterator.

Recall that the **for** can cycle through any collection of objects that implement the **Iterable** interface. Because all of the collection classes implement this interface, they can all be operated upon by the **for**.

**package** collection;

**import** java.util.ArrayList;

**class** ForEachDemo {

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

ArrayList<Integer> vals = **new** ArrayList<Integer>();

vals.add(1);

vals.add(2);

vals.add(3);

vals.add(4);

vals.add(5);

System.***out***.print("Contents of vals: ");

**for** (**int** v : vals)

System.***out***.print(v + " ");

System.***out***.println();

**int** sum = 0;

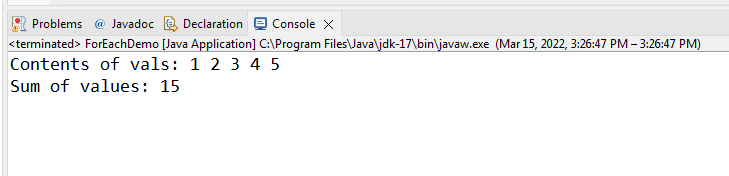
**for** (**int** v : vals)

sum += v;

System.***out***.println("Sum of values: " + sum);

}

}



As you can see, the **for** loop is substantially shorter and simpler to use than the iterator-based approach. However, it can only be used to cycle through a collection in the forward direction, and you can’t modify the contents of the collection.

# Java Vector

**Vector** is like the dynamic array which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.

It is similar to the ArrayList, but it consumes more memory than ArrayList

**Empty vector has initial size of 10**

**import** java.util.Vector;

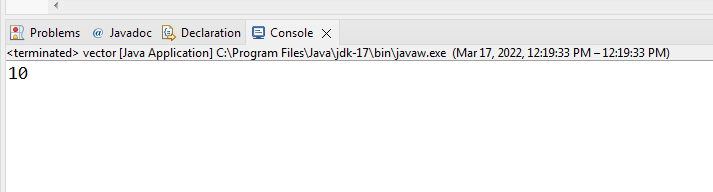
**class** vector {

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

Vector<Integer> s = **new** Vector<>();

System.***out***.println(s.capacity());

}}



**import** java.util.Vector;

**class** vector {

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

Vector<Integer> s = **new** Vector<>();

System.***out***.println(s.capacity());

s.add(66);

s.add(66);

s.add(66);

s.add(66);

s.add(66);

s.add(66);

s.add(66);

s.add(66);

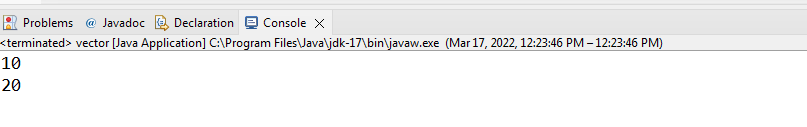
s.add(66);

s.add(66);

s.add(66);//size will be double when you insert 11th element

System.***out***.println(s.capacity());

}}



**Empty arraylist has initial size of 0**

**import** java.util.ArrayList;

**class** vector {

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

ArrayList<Integer> s = **new** ArrayList<>();

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

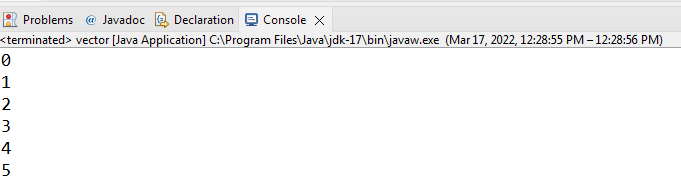
s.add(66);

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

}}



# Java LinkedList class

Java LinkedList class uses a doubly linked list to store the elements.ArrayList and Vector uses Dynamic Array to store elements It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.(when we insert/remove elements in between shifting happens upwards/downwards)
* Java LinkedList class can be used as a list, stack or queue.

### Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

**import** java.util.LinkedList;

**class** vector {

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

LinkedList<Integer> s = **new** LinkedList<>();

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

s.add(66);

System.***out***.println(s.size());

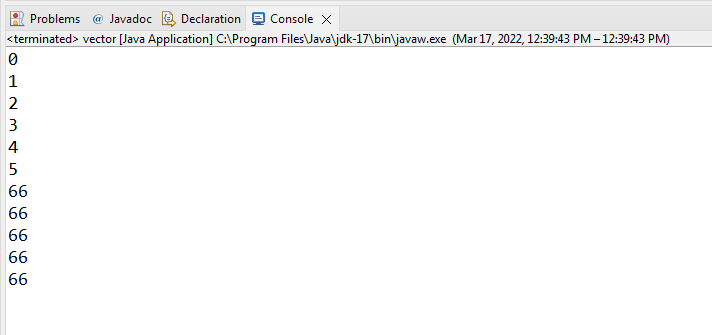
s.add(66);

System.***out***.println(s.size());

**for** (**int** i : s) {

System.***out***.println(i);

}}}



# Set in Java

The **set** is an interface available in the **java.util** package. The **set** interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a **collection interface**. The set interface is used to create the mathematical set. The set interface use collection interface's methods to avoid the insertion of the same elements. **SortedSet** and **NavigableSet** are two interfaces that extend the set implementation.

**Duplicates are not allowed in Set**

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** hashset {

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

Set<Integer> s = **new** HashSet<>();

s.add(66);

s.add(66);

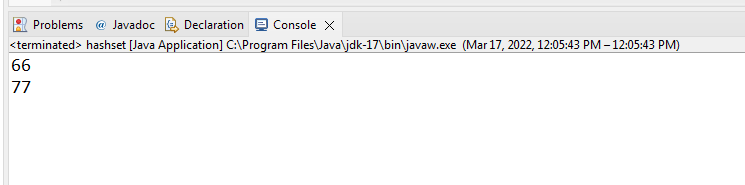
s.add(77);

s.add(66);

**for** (**int** i : s) {

System.***out***.println(i);

} }}



**Set doesn’t maintain sequence**

HashSet follows hashing in which your values are storing inside the heap and hashing used some algorithm to fetch nearest value in heap

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** hashset {

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

Set<Integer> s = **new** HashSet<>();

s.add(66);

s.add(55);

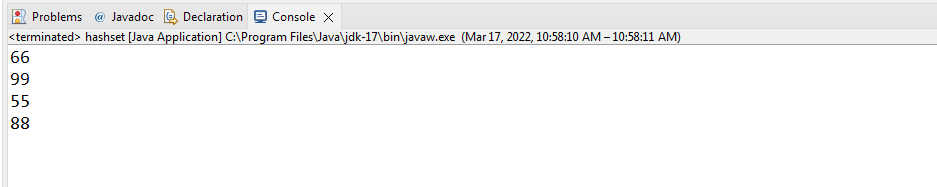
s.add(88);

s.add(99);

**for** (**int** i : s) {

System.***out***.println(i);

} }}



**If you want to maintain sequence in ascending order than use TreeSet**

**import** java.util.Set;

**import** java.util.TreeSet;

**public** **class** treeset {

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

Set<Integer> s = **new** TreeSet<>();

s.add(66);

s.add(55);

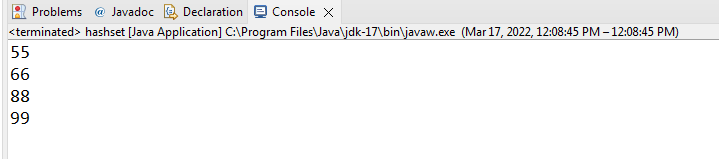
s.add(88);

s.add(99);

**for** (**int** i : s) {

System.***out***.println(i);

} }}



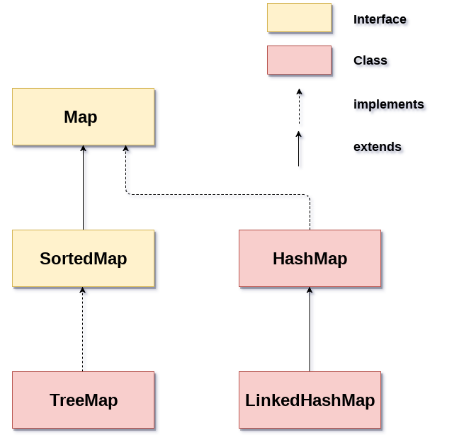
# Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

## Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

**HashMap:doesn’t maintain sequence like HashSet**

**import** java.util.HashMap;

**import** java.util.Map;

**class** map {

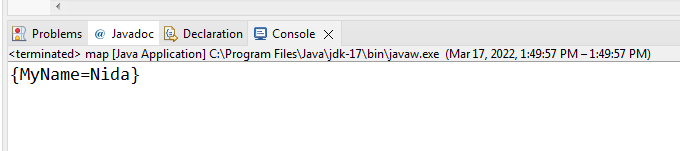
**public** **static** **void** main(String[] args) {

Map<String, String> s = **new** HashMap<>();

s.put("MyName" , "Nida");

System.***out***.println(s);

}}



**HashMap:doesn’t maintain sequence like HashSet**

**import** java.util.HashMap;

**import** java.util.Map;

**class** map {

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

Map<String, String> s = **new** HashMap<>();

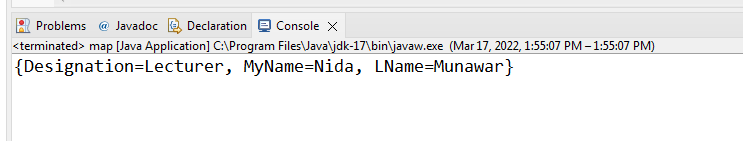
s.put("MyName" , "Nida");

s.put("LName" , "Munawar");

s.put("Designation" , "Lecturer");

System.***out***.println(s);

}}



**Search with key**



**import** java.util.HashMap;

**import** java.util.Map;

**class** map {

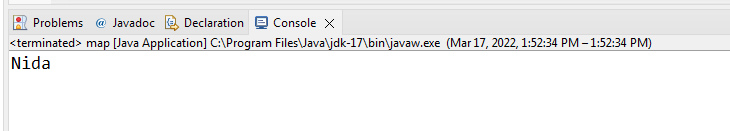
**public** **static** **void** main(String[] args) {

Map<String, String> s = **new** HashMap<>();

s.put("MyName" , "Nida");

System.***out***.println(s.get("MyName"));

}}



**For any key which is not available it will give you null**

**import** java.util.HashMap;

**import** java.util.Map;

**class** map {

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

Map<String, String> s = **new** HashMap<>();

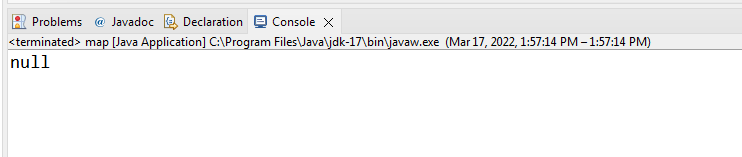
s.put("MyName" , "Nida");

s.put("LName" , "Munawar");

s.put("Designation" , "Lecturer");

System.***out***.println(s.get("SCD"));

}}



**We have 2 ways for Printing all keys and values**

**1.using For each loop**

**We are using methods of Set to store keys**



**keyset() will return all the keys present in the map**

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**class** map {

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

Map<String, String> s = **new** HashMap<>();

s.put("MyName" , "Nida");

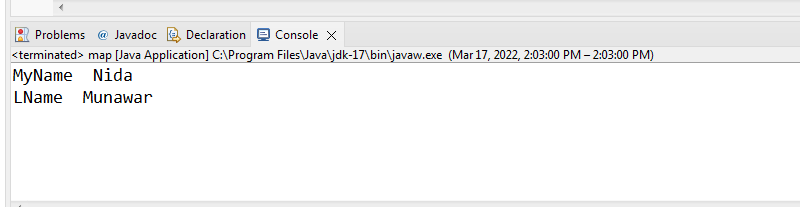
s.put("LName" , "Munawar");

Set<String> s1 = s.keySet();

**for**(String k : s1)

System.***out***.println(k + " " + s.get(k));

}}



## 2.Map.Entry Interface

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

The **entrySet( )** method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

**What is entry?**

Key and Value pair makes one entry

**import** java.util.HashMap;

**import** java.util.Map;

**class** map {

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

Map<Integer,String> map=**new** HashMap();

map.put(1,"Nida");

map.put(2,"Bakhtawar");

map.put(3,"Atiya");

map.put(4,"Romasha");

//Traversing Map

**for**(Map.Entry m:map.entrySet()){

System.***out***.println(m.getKey()+" "+m.getValue());

}

}}

**Or you can use set to store keys and values**

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**class** map {

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

Map<Integer,String> map=**new** HashMap();

map.put(1,"Nida");

map.put(2,"Bakhtawar");

map.put(3,"Atiya");

map.put(4,"Romasha");

Set<Map.Entry<Integer,String>> pair = map.entrySet();

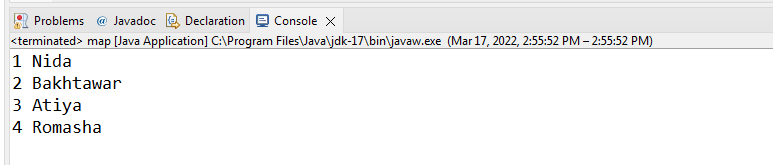
//Traversing Map

**for**(Map.Entry m:pair){

System.***out***.println(m.getKey()+" "+m.getValue());

}

}}



**We cannot repeat/duplicate keys**

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**class** map {

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

Map<String, String> s = **new** HashMap<>();

s.put("MyName" , "Nida");

s.put("LName" , "Munawar");

s.put("MyName" , "Atiya");

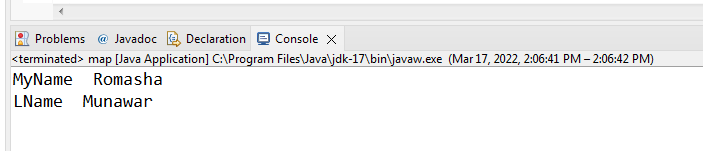
s.put("MyName" , "Romasha");

Set<String> s1 = s.keySet();

**for**(String k : s1)

System.***out***.println(k + " " + s.get(k));

}}



**We can repeat/duplicate values**

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**class** map {

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

Map<String, String> s = **new** HashMap<>();

s.put("MyName" , "Nida");

s.put("LName" , "Munawar");

s.put("MyName" , "Nida");

s.put("Name" , "Nida");

Set<String> s1 = s.keySet();

**for**(String k : s1)

System.***out***.println(k + " " + s.get(k));

}}

