Generics in Java

Java Generics allows us to create a single class, [interface](https://www.programiz.com/java-programming/interfaces), and [method](https://www.programiz.com/java-programming/methods) that can be used with different types of data (objects). This helps in reusing the code.

Let us understand why we need Generics.

Say, we want to create a class that prints the passed integer in the console.

We will create a class called IntegerPrinter and add the below code

**Public class** IntegerPrinter(){

**Integer** thingsToPrint;

**public** IntegerPrinter(Integer print){

**this.** thingsToPrint = print;

}

**Public** void printMethod(){

System.out.println(thingsToPrint);

}}

Now, If we have to do the same thing for double or string, we have to copy the class for double and String. This will lead to code duplication.

Another way of doing this is using generics.

**public** **class** Printer<T> {

T thingsToPrint;

**public** Printer(T print){

**this**. thingsToPrint = print;

}

**public** **void** printMethod(){

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

}

}

-------------Main-------------

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

Printer<Integer> printer = **new** Printer<>(23);

printer.printMethod();

}

Note – Generics does not work with primitive datatypes

Java Collections

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

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

A Collection represents a single unit of objects, i.e., a group. Here framework refers to readymade architecture.

Java Collection has the following

* List collection
* Set Collection
* Map collection

# List Collection

It is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collections of the objects.

List collection has the following:

1. ArrayList
2. LinkedList
3. Vector

Note :- List is an interface and the above three are classes implementing the method of List interface.

## ArrayList

ArrayList provides us with dynamic arrays in Java. The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection. ArrayList is a Java class implemented using the List interface.

Using an array list:

ArrayList<Integer> arr2 = **new** ArrayList<Integer>();

//adding elements to arrayList

arr2.add(i);

### Java ArrayList Methods

|  |  |
| --- | --- |
| Methods | Description |
| add(int i) | To add elements to attaylist at the mentioned index |
| clear() | To remove all the elements |
| contains() | To check if an arraylist contains a particular element |
| get(int i) | Gets the element at index i |
| indexOf(object) | Gets the index of object/element mentioned |
| isEmpty() | Returns true if arrayList is empty |
| remove(int i) | Removes the element at index i |
| size() | Returns the number of elements in an array |
| set(index, value) | Sets the element value at the specified index |
| Collections.sort(list); | Used for sorting the arrayList |

## LinkedList

Linkedlist is also implements List interface. In LinkedList the data is stored with the pointers where we have data and the address of where the data is stored. In LinkedList data is not stored in continues order instead they are stored in different memory locations.

LinkedList<String> ll = **new** LinkedList<>();

Difference between linkedlist and arraylist

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| ArrayList internally uses a dynamic array to store the elements. | LinkedList internally uses a doubly linked list to store the elements. |
| Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| ArrayList is better for storing and accessing data. | LinkedList is **better for manipulating** data. |
| The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |

### Methods of Linkedlist

|  |  |
| --- | --- |
| Methods | Description |
| add(int i)  addFirst/addLast | To add elements to list at the mentioned index |
| clear() | To remove all the elements |
| contains() | To check if an list contains a particular element |
| get(int i) | Gets the element at index i |
| indexOf(object) | Gets the index of object/element mentioned |
| remove(int i)  removeFirst/removeLast | Removes the element at index i |
| size() | Returns the number of elements in an array |
| set(index, value) | Sets the element value at the specified index |
| Collections.sort(list); | Used for sorting the List |

## Vectors

* [Vector](https://www.geeksforgeeks.org/java-util-vector-class-java/) provides us with dynamic arrays in Java. This is a legacy class. Vector implements a dynamic array which means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index.
* It also maintains an insertion order like an ArrayList. Still, it is rarely used in a non-thread environment as it is synchronized, and due to this, it gives a poor performance in adding, searching, deleting, and updating its elements.

Vector<Integer> v2 = **new** Vector<Integer>();

# Set Interface

Set Interface has HashSet and TreeSet. Itis an unordered collection of objects in which duplicate values cannot be stored.

## HashSet

* Implements [Set Interface](https://www.geeksforgeeks.org/set-in-java/).
* As it implements the Set Interface, duplicate values are not allowed.
* Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.
* NULL elements are allowed in HashSet.

HashSet<String> h = **new** HashSet<String>();

* Difference between Set and List is that List stores data in indexed sequence where as in set it is not indexed.
* List allows duplicate values while Set doesnot allow duplicate values.
* We can remove or add elements in list based on index which is not possible in Set.

### Methods in HashSet

|  |  |
| --- | --- |
| Methods | Description |
| add(Object e) | To add elements to HashSet |
| clear() | To remove all the elements |
| contains() | To check if an arraylist contains a particular element |
| remove(Object e) | Removes the element |
| size() | Returns the number of elements in an array |
| isEmpty() | Returns true if set is empty |

## TreeSet

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in a sorted and ascending order.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

### Methods in Treeset

|  |  |
| --- | --- |
| Methods | Description |
| add(Object e) | To add elements to HashSet |
| clear() | To remove all the elements |
| contains() | To check if an arraylist contains a particular element |
| remove(Object e) | Removes the element |
| size() | Returns the number of elements in an array |
| isEmpty() | Returns true if set is empty |
| first()/last() | Returns first/last elements in set |

Example:

TreeSet ts = new TreeSet();

// Add elements to the tree set

ts.add("C");

ts.add("A");

ts.add("B");

ts.add("E");

ts.add("F");

ts.add("D");

System.out.println(ts);

# Map Interface

Map is a data structure that supports the key-value pair mapping for the data. It is mainly used in the scenarios where Students are needed to be searched or removed or added according to the roll numbers.

## HashTable

* The **Hashtable** class implements a hash table, which maps keys to values.
* The **java.util.Hashtable** class is a class in Java that provides a **key-value** data structure, similar to the Map interface.
* It was part of original Java collections introduced in Java 1.0
* The `**Hashtable`**class has since been considered obsolete and its use is generally discouraged. Due its synchronization nature.

## HashMap

* [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) uses a technique called Hashing.
* It stores the data in (Key, Value) pairs.
* To access a value in a HashMap, we must know its key.
* It allows to store the null keys as well, but there should be only one null key object and there can be any number of null values.

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

### Methods in Hashmap

|  |  |
| --- | --- |
| Methods | Description |
| clear() | Clears the map |
| containsKey() | Returns true if the mentioned key is present |
| containsValue() | Returns true if the mentioned value is present |
| get(Key) | Returns the value for the mentioned key |
| isEmpty() | Returns true if map is empty |
| keySet() | Returns set of keys in the map |
| put(key,value) | Used to add key-values to map |
| remove(key) | Used to remove key-value from map |
| size() | Returns the size of the map |
| values() | Returns a set of values in the map |

## TreeMap

* Java TreeMap contains values based on the key.
* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap maintains ascending order.

Map<String, Integer> treeMap = **new** TreeMap<>();

|  |  |
| --- | --- |
| Methods | Description |
| clear() | Clears the map |
| containsKey() | Returns true if the mentioned key is present |
| containsValue() | Returns true if the mentioned value is present |
| get(Key) | Returns the value for the mentioned key |
| keySet() | Returns set of keys in the map |
| put(key,value) | Used to add key-values to map |
| remove(key) | Used to remove key-value from map |
| size() | Returns the size of the map |
| values() | Returns a set of values in the map |

# Comparable and Comparator

* Comparable and Comparator both are interfaces and can be used to sort collection elements.
* A comparable object is capable of comparing itself with another object.
* Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.
* **java.lang.Comparable** and **java.util.Comparator** interfaces to sort array/list of custom classes.

Implementing Comparable:

Create a class with Students name and age:

**class** Students **implements** Comparable<Students>

{

**int** age;

String name;

Students(**int** age, String name)

{

**this**.age = age;

**this**.name = name;

}

@Override

**public** **int** compareTo(Students o2) {

**if**(**this**.age > o2.age)

**return** 1;

**else**

**return** -1;

}

}

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

List<Students> studs = **new** ArrayList<>();

studs.add(**new** Students(21, "John"));

studs.add(**new** Students(18, "Peter"));

studs.add(**new** Students(25, "Kelly"));

studs.add(**new** Students(19, "Mark"));

Collections.*sort*(studs);

**for**(Students s:studs)

{

System.***out***.println(s.age+" "+s.name);

}

}

Implementing Comparator:

Comparator uses Lambda expression. The below line can be added in main:

Comparator<Students> comp = (i,j) -> i.age>j.age?1:-1;

For sorting

Collections.*sort*(studs, comp);

Same way we can use below line to sort based on students names:

Comparator<Students> comp = (i,j) -> i.name.compareTo(j.name);

# Collections Utility class

|  |  |
| --- | --- |
| **Methods** | **Description** |
| Collections.Sort(List myList) | Sorts the given list (myList) in natural ordering |
| Collections.shuffle(List myList) | Puts the elements in myList in random order |
| Collections.reverse(List myList) | Reverses the elements of myList |
| Collections.binarySearch(List mlist, T key) | Performs binary search on the mentioned key |
| Collections.copy(List dest, List src) | Copies elements from one collection to another |
| Collections.frequency(Collection c, Object o) | Returns the number of elements in the specified collection class c equal to the specified object |

# Stream API

* Steam API was introduced in Java 8
* Stream API is used to process collections of objects.
* Enable us to perform operations like filtering, mapping, counting, and sorting.

Let us do some operation using steam APIs

Let’s say we have do some operation on the below list:

List<Integer> nums = Arrays.*asList*(4,3,5,1,8,9);

We can create a steam API for this which can filter, sort, map the data.

Steams will not modify the existing data. Instead it will creates a stream of data and works on it.

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

List<Integer> nums = Arrays.*asList*(4,3,5,1,8,9);

Stream<Integer> newStream = nums.stream();

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

newStream.filter(n -> n%2 ==0).forEach(n -> System.***out***.println(n));

newStream.map(n -> n\*2).forEach(n -> System.***out***.println(n));

newStream.forEach(n -> System.***out***.println(n));

}

Exercise

* Take a list of String and print names starting with S
* Print names containing 4 characters