Generics

class StudentRecord {

Student[] stud = new Student[10];

public void add(Student s) {

//logic

}

public Student get(int rollno) {

//logic

}

}

class BookRecord {

Book[] books = new Book[10];

public void add(Book b) {

//logic

}

public Book get(int bookid) {

//logic

}

}

* In the above 2 classes are created for the same operations but to perform on different object types.
* The first class is performing operation on Student type objects and the second class is performing operation on Book type objects.
* if we have more object types, then we need to create more number of classes, for the same operations.
* It will increase the number of classes in a project.
* So, the solution is, create a generic class.
* A generic class is a class, but it can be used with multiple object types, to perform the same operations.
* A generic class, is a class with type parameters.

for example:

class Record<T> {

T[] arr = new T[10];

public void add(T t) {

//logic

}

public T get(int id) {

//logic

}

}

* while creating a generic class/ generic interface, the classname/interface should contain type parameters with in angle brackets.
* The parameter types can be,

The most commonly used type parameter names are:

* E - Element (used extensively by the Java Collections Framework)
* K - Key
* N - Number
* T - Type
* V - Value
* S,U,V etc. - 2nd, 3rd, 4th types
* object creation for the above generic class is

like below.

Record<Book> bookRecord = new Record<Book>(); Record<Student> studentRecord=new Record<Student>();

Collection framework

. Data structure is a concept, which works on a principle, to organize the data efficiently, so that storing the data, retrieving the data, searching for the data becomes easy and fast.

* The principles are,

LIFO : ex: undo operation.

FIFO : ex: printer jobs

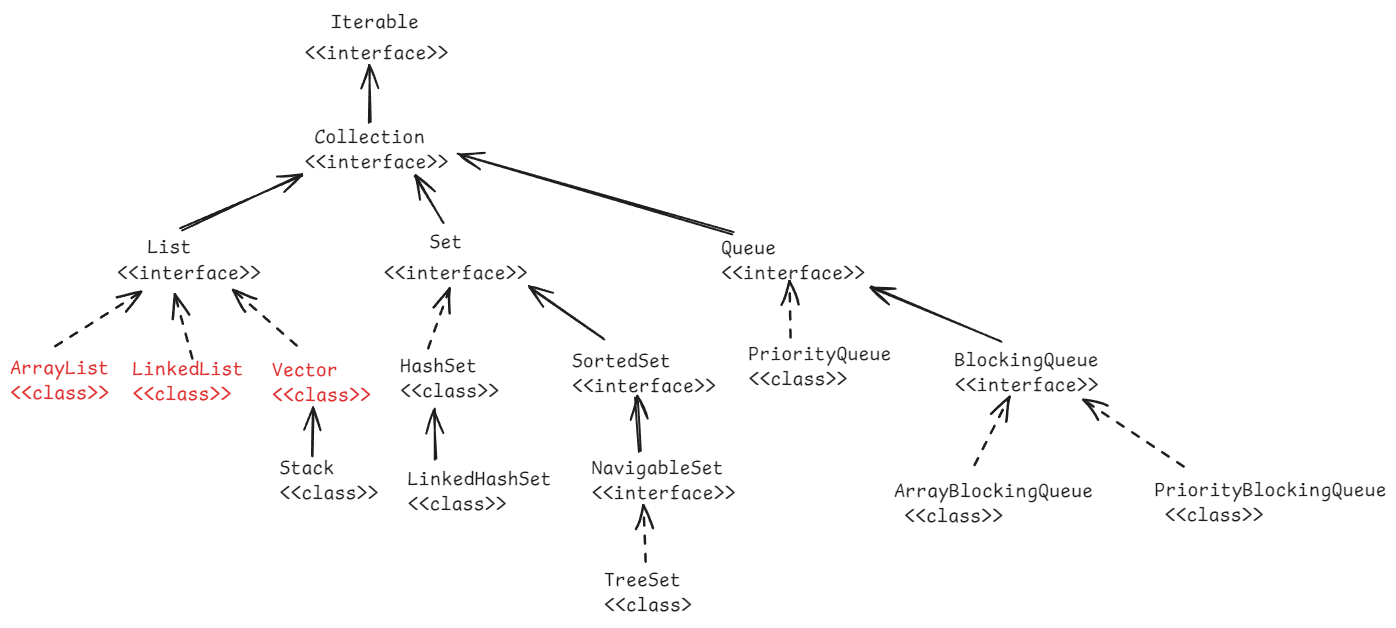
Linked list : ex: browsing history

Tree: ex: company employees hierachy

Graph: ex: route map

In Java, the language has provided a set of pre-defined classes and interfaces by implementing Data structure principles, and these classes and interfaces is combinedly called Collection Framework.

Collection interface hierarchy:



Q) what is the difference between array and a collection?

A) 1. array can store values of same type.(homogeneous)

collection can store objects of either same type

or different type. ( hetrogeneous)

2. in array, size is fixed. But in collection size is not fixed.

methods of Collection interface:

1. add(E e) : adds the given element to the collection
2. addAll(Collection c) : adds all the elements of the given collection to this collection.
3. clear(): removes all the elements from this collection.
4. contains(Object o): returns true, if the given element exist in this collection. Otherwise, returns false.
5. isEmpty(): returns true, if this collection is empty. Otherwise, returns false.
6. iterator(): returns an iterator over this collection.
7. remove(Object o): removes the given element from this collection.
8. size() : returns the number of elements in this collection.
9. toArray(): converts this collection to an Object array.

Methods of List interface:

1. add(index, element): Inserts the specified element at the specified position in this list.
2. get(index): Returns the element at the specified position in this list.
3. indexOf(Object o) : Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
4. lastIndexOf(Object o): Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
5. listIterator(): returns a list iterator over the elements of this list.
6. remove(index): removes the element from the given index.
7. set(index, element): Replaces the element at the specified position in this list with the specified element.
8. sort(Comparator c): Sorts the elements of this list based on the given Comparator.
9. subList(startIndex, endIndex): returns a list of elements from startIndex to endIndex-1.

====================================================

equals() and hashCode() :

* equals() method belongs to java.lang.Object class.
* The equals() method of Object class works like == operator.
* If you want to compare the two objects of your class with content, then you have to override equals() method of Object class.

ex1:

Without overriding equals() method.

**package** pack1;

**class** A {

**int** x;

A(**int** x) {

**this**.x = x;

}

}

**public** **class** Main {

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

A a1 = **new** A(10);

A a2 = **new** A(10);

System.***out***.println(a1.equals(a2)); //false

}

}

ex2: with overriding equals() method.

**package** pack1;

**class** A {

**int** x;

**int** y;

A(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

A other = (A) obj;

**return** x == other.x && y == other.y;

}

}

**public** **class** Main {

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

A a1 = **new** A(10, 20);

A a2 = **new** A(10, 20);

System.***out***.println(a1.equals(a2)); //true

}

}

hashCode():

* hashCode() is a method of Object class.
* hash code is an integer value.
* if you don’t override hashCode() method in your class then Object class hashCode() method is called.
* Object class hashCode() returns an integer by converting the memory address of the object.
* if you want to get same hash code for the objects, if they are equal with respect to equals() (means, content), then you have to override hashCode() method in your class.

ex1: without overriding hashCode() method.

**package** pack1;

**class** A {

**int** x;

**int** y;

A(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

A other = (A) obj;

**return** x == other.x && y == other.y;

}

}

**public** **class** Main {

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

A a1 = **new** A(10, 20);

A a2 = **new** A(10, 20);

System.***out***.println(a1.hashCode());

System.***out***.println(a2.hashCode());

}

}

output:

1175962212

918221580

ex2: with overriding hashCode() method.

**package** pack1;

**import** java.util.Objects;

**class** A {

**int** x;

**int** y;

A(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(x, y);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

A other = (A) obj;

**return** x == other.x && y == other.y;

}

}

**public** **class** Main {

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

A a1 = **new** A(10, 20);

A a2 = **new** A(10, 20);

System.***out***.println(a1.hashCode());

System.***out***.println(a2.hashCode());

}

}

output:

1291

1291

toString() method:

* toString() method belongs to java.lang.Object class.
* If you want to convert an object to a string format then you have to override toString() method.
* If you don’t override toString() method then Object class toString() method is called and it returns a string in the below format.

Fully qualified classname@unsigned hexa decimal form of object hash code.

**package** com.ashokit.example;

**class** ClassA {

**private** **int** x;

**private** **int** y;

ClassA() {

x = 100;

y = 200;

}

@Override

**public** String toString() {

**return** "ClassA [x=" + x + ", y=" + y + "]";

}

}

**public** **class** MainClass {

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

ClassA ca = **new** ClassA();

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

}

}

ArrayList class:

----------------

* If you want to perform CRUD operations on a group of objects/elements, by allowing duplicate elements then you have to create ArrayList object.
* ArrayList class internally uses resizable array, to store the elements.
* If we create an ArrayList class object, with out initial capacity, then by default the capacity is 10.
* When all the elements are filled then its capacity will be increased by 50%.
* Suppose, if initial capacity is 10, and while adding the 11th element, the capacity will be increased to 15.
* We can create ArrayList object in either normal way or in a generic way.

ArrayList lst1 = new ArrayList(); //normal way.

* + This object can store heterogeneous elements.

ArrayList<Employee> lst2 = new ArrayList<Employee>(); //generic way

* + This object can store homogeneous elements.

Note:

While creating the object in generic way, specifying the type parameter at left side of object creation is mandatory, but at right side of object creation is optional.

For example,

ArrayList<Employee> lst = new ArrayList<Employee>();

(or)

ArrayList<Employee> lst = new ArrayList<>();

Example1:

**package** pack1;

**import** java.util.ArrayList;

**public** **class** MainClass {

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

ArrayList<Integer> lst1 = **new** ArrayList<>();

lst1.add(10);

lst1.add(20);

lst1.add(30);

lst1.add(2, 40); //inserting

System.***out***.println("array list elements, after inserting.");

System.***out***.println(lst1);

lst1.set(3, 50); //replacing

System.***out***.println("array list elements, after replacing");

System.***out***.println(lst1);

lst1.remove(2);

System.***out***.println("array list elements, after removing");

System.***out***.println(lst1);

lst1.clear();

System.***out***.println("array list elements, after clear");

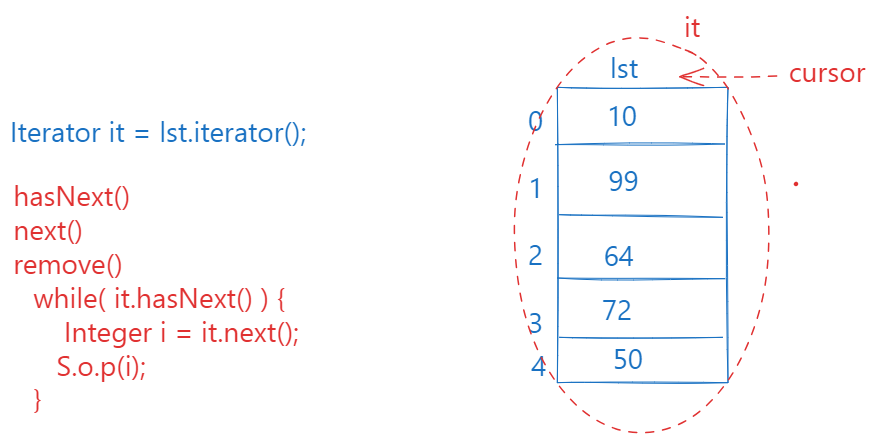
System.***out***.println(lst1);

}

}

iterator() method:

* Iterating a collection object is nothing but, visiting the elements of a collection object.
* This iterator() method creates, an Iterator object, through which, we can iterate a collection.
* Iterator object, internally maintains a cursor, and it is by default placed at before the first element.
* Iterator is an interface and it has the below methods.
  + 1. hasNext()
    2. next()
    3. remove()



for example,

ArrayList<Integer> lst1 = **new** ArrayList<>();

lst1.add(10);

lst1.add(20);

lst1.add(30);

lst1.add(2, 40); //inserting

Iterator<Integer> it = lst1.iterator();

**while**( it.hasNext() ) {

Integer i = it.next();

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

}

* we can also, iterate/visit the elements of a collection using for each loop.
* Ex:

ArrayList<Integer> lst1 = **new** ArrayList<>();

lst1.add(10);

lst1.add(20);

lst1.add(30);

lst1.add(2, 40); //inserting

**for** ( Integer i : lst1 ) {

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

}

/\*

\* create a class Employee with attributes empno, ename, sal, gender and experience

\* In main class, create a list of employees, then display only female

\* employees and also display only the employees with experience < 5 years.

\*/

**package** pack1;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

}

**public** **class** MainClass {

**private** **static** ArrayList<Employee> getEmployeesList() {

ArrayList<Employee> empList = **new** ArrayList<>();

empList.add(**new** Employee(7298, "Scott", 5000.0, "Male", 4.5));

empList.add(**new** Employee(7178, "Allen", 7000.0, "Male", 5.5));

empList.add(**new** Employee(7154, "Kathey", 6000.0, "Female", 4.5));

empList.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

empList.add(**new** Employee(7741, "Mary", 4000.0, "Female", 3.1));

empList.add(**new** Employee(7865, "Daniel", 9000.0, "Male", 7.5));

empList.add(**new** Employee(7539, "Jennie", 7000.0, "Female", 5.5));

empList.add(**new** Employee(7932, "Annie", 3000.0, "Female", 2.5));

empList.add(**new** Employee(7691, "John", 4000.0, "Male", 3.5));

empList.add(**new** Employee(7356, "Lisa", 9000.0, "Female", 6.5));

**return** empList;

}

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

ArrayList<Employee> empList = *getEmployeesList*();

Iterator<Employee> it = empList.iterator();

System.***out***.println("Displaying female employees...........");

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

**while**(it.hasNext()) {

Employee emp = it.next();

**if**(emp.getGender().equalsIgnoreCase("Female")) {

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

}

}

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

System.***out***.println("Displaying employees with experience < 5 years");

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

Iterator<Employee> it2 = empList.iterator();

**while**(it2.hasNext()) {

Employee emp = it2.next();

**if**(emp.getExperience() < 5) {

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

}

}

}

}

Comparable<T> interface:

* It is a functional interface, with a single abstract method(sam) called, compareTo().
* If you want to sort the objects of a class in natural sorting order(ascending order), then you have to implement your class from Comparable interface.

for ex:

class Employee implements Comparable<Employee> {

//variables

//constructors

//getters and setters

@Override

public int compareTo(Employee o) {

//logic

}

}

compareTo() method returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

* Comparable interface should be implemented into a class, whose objects you want to sort.
* Suppose, if you want to sort Customer objects then you have to implement Comparable interface into Customer class.
* ex:
* class Customer implements Comparable<Customer> {
* //variables
* //constructors&methods
* //override compareTo() method
* }
* The limitation of Comparable<T> interface is, at a time we can sort the objects of a class on a single attribute/property.

/\*

\* write a program to sort the list of employees

\* in ascending order of their names using comparable interface

\*/

**package** pack1;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.List;

**class** Employee **implements** Comparable<Employee>{

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** compareTo(Employee o) {

**return** **this**.getEname().compareTo(o.getEname());

}

}

**public** **class** Main {

**private** **static** ArrayList<Employee> getEmployeesList() {

ArrayList<Employee> empList = **new** ArrayList<>();

empList.add(**new** Employee(7298, "Scott", 5000.0, "Male", 4.5));

empList.add(**new** Employee(7178, "Allen", 7000.0, "Male", 5.5));

empList.add(**new** Employee(7154, "Kathey", 6000.0, "Female", 4.5));

empList.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

empList.add(**new** Employee(7741, "Mary", 4000.0, "Female", 3.1));

empList.add(**new** Employee(7865, "Daniel", 9000.0, "Male", 7.5));

empList.add(**new** Employee(7539, "Jennie", 7000.0, "Female", 5.5));

empList.add(**new** Employee(7932, "Annie", 3000.0, "Female", 2.5));

empList.add(**new** Employee(7691, "John", 4000.0, "Male", 3.5));

empList.add(**new** Employee(7356, "Lisa", 9000.0, "Female", 6.5));

**return** empList;

}

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

List<Employee> lstEmp = *getEmployeesList*();

Collections.*sort*(lstEmp);

**for**(Employee e : lstEmp) {

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

}

}

}

Comparator<T> interface:

* It is a functional interface with a single abstract method(sam), called compare() method.
* We have to create a separate class by implementing Comparator interface, for comparing the objects of a type.
* we can also created multiple comparator classes, if you need to sort the elements of your collection on different properties.
* for ex:

**class** EmpnoComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getEmpno() - o2.getEmpno();

}

}

**class** SalaryComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** (**int**) (o1.getSal() - o2.getSal());

}

}

/\*

\* write a program to sort the list of employees

\* in ascending order of their numbers and also

\* their salaries using Comparator interface

\*/

**package** pack1;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.List;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

}

**class** EmpnoComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getEmpno() - o2.getEmpno();

}

}

**class** SalaryComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** (**int**) (o1.getSal() - o2.getSal());

}

}

**public** **class** Main {

**private** **static** ArrayList<Employee> getEmployeesList() {

ArrayList<Employee> empList = **new** ArrayList<>();

empList.add(**new** Employee(7298, "Scott", 5000.0, "Male", 4.5));

empList.add(**new** Employee(7178, "Allen", 7000.0, "Male", 5.5));

empList.add(**new** Employee(7154, "Kathey", 6000.0, "Female", 4.5));

empList.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

empList.add(**new** Employee(7741, "Mary", 4000.0, "Female", 3.1));

empList.add(**new** Employee(7865, "Daniel", 9000.0, "Male", 7.5));

empList.add(**new** Employee(7539, "Jennie", 7000.0, "Female", 5.5));

empList.add(**new** Employee(7932, "Annie", 3000.0, "Female", 2.5));

empList.add(**new** Employee(7691, "John", 4000.0, "Male", 3.5));

empList.add(**new** Employee(7356, "Lisa", 9000.0, "Female", 6.5));

**return** empList;

}

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

List<Employee> lstEmp = *getEmployeesList*();

Collections.*sort*(lstEmp, **new** EmpnoComparator());

System.***out***.println("List of Employees sorted in empno order");

**for** (Employee e : lstEmp) {

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

}

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

Collections.*sort*(lstEmp, **new** SalaryComparator());

System.***out***.println("List of Employees sorted in salary order");

**for** (Employee e : lstEmp) {

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

}

}

}

Q) what is the difference between Comparable and Comparator?

Q) is an ArrayList object, a thread-safe?

A) No, it is not a thread-safe object.

Q) Can we make ArrayList object as thread-safe?

A) Yes. We can create a synchronized array list object by calling synchronizedList() method of Collections class.

ex:

List<Employee> newLstEmp = Collections.synchronizedList(lstEmp);

Q) is an ArrayList object, a mutable?

A) Yes. It is mutable object.

Q) can we make ArrayList object as immutable?

A) Yes. We can create immutable array list object, by calling unmodifiableList() method of Collections class.

ex:

List<Employee> newLstEmp = Collections.unmodifiableList(lstEmp);

Q) When to use ArrayList and when to use LinkedList?

A) If read operation is required more times and write operation less times then use ArrayList object.

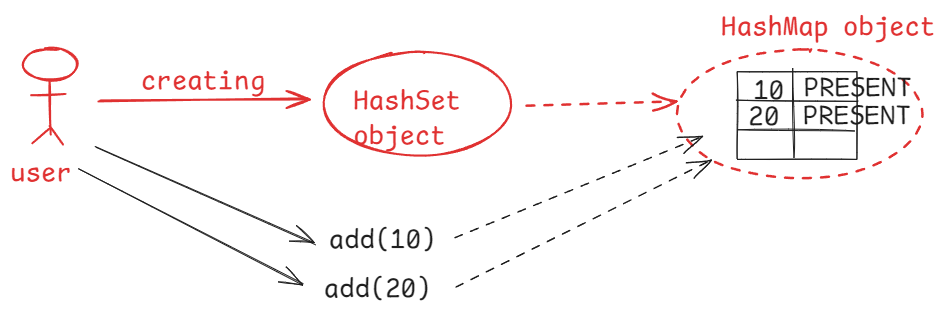
If write operation is required more times and read opertion less times then use LinkedList object.

HW: learn Vector class and go through the LinkedList class that you have already learnt.

HashSet class:

* HashSet class is an implementation of Set interface.
* HashSet class internally uses HashMap class.
* When HashSet class object is created, then internally HashMap class object is created.
* The HashMap object follows hash table data structure, and this table structures stores elements in key-value pairs.
* When a HashSet object is created with parameter less constructor, internally HashMap object is created with default initial capacity as 16 and the load factor as 0.75.
* when you add an element to the HashSet object, internally that element will be stored as a key with value PRESENT, in the map object.
* Here, PRESENT is an object of Object class.
* // Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object ***PRESENT*** = **new** Object();



* The internal table structure is doubled, when the number of elements are reached to

capacity \* load factor.

* for example, if capacity is 16 and load factor 0.75, then 16 \* 0.75 = 12 elements, now the internal table structure capacity is doubled to 32.
* when we add an element to the HashSet object, internally the hash code of the element is calculated, next the bucket index is calculated with formula hashCode(element) & (n-1) then the element will be inserted into that bucket as a key with the dummy value PRESENT.
* As Set interface extends Collection interface, it has Collection interface methods and as HashSet class implements Set interface, the HashSet class has the same methods.
* Set interface does not added any additional methods, and HashSet class added a single new method clone().

HashSet<String> hs = new HashSet<>();

. The internal HashMap object is created with default initial capacity as 16, with load factor 0.75.

HashSet<String> hs = new HashSet<>(10, 0.6f);

. The internal HashMap object is created with default initial capacity as 10, with load factor 0.6.

HashSet<String> hs = new HashSet<>(10);

. The internal HashMap object is created with default initial capacity as 10, with load factor 0.75.

Q) what if, I add a duplicate element to the HashSet?

A) The add() method returns a boolean value. If the element is already exist in the HashSet object then

add() returns false. But we will not get any compile-time error or an exception at runtime.

Q) what if, I remove an element which is not exist in the HashSet?

A) The remove() method returns a boolean value. If the element doesn’t exist then remove() method returns false. But we will not get any error or exception.

**package** p1;

**import** java.util.HashSet;

**import** java.util.Objects;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(empno, ename, experience, gender, sal);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**return** empno == other.empno && Objects.*equals*(ename, other.ename)

&& Double.*doubleToLongBits*(experience) == Double.*doubleToLongBits*(other.experience)

&& Objects.*equals*(gender, other.gender)

&& Double.*doubleToLongBits*(sal) == Double.*doubleToLongBits*(other.sal);

}

}

**public** **class** Solution {

**private** **static** HashSet<Employee> getEmployeesSet() {

HashSet<Employee> empSet = **new** HashSet<>();

empSet.add(**new** Employee(7298, "Scott", 5000.0, "Male", 4.5));

empSet.add(**new** Employee(7178, "Allen", 7000.0, "Male", 5.5));

empSet.add(**new** Employee(7154, "Kathey", 6000.0, "Female", 4.5));

empSet.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

empSet.add(**new** Employee(7741, "Mary", 4000.0, "Female", 3.1));

empSet.add(**new** Employee(7865, "Daniel", 9000.0, "Male", 7.5));

empSet.add(**new** Employee(7539, "Jennie", 7000.0, "Female", 5.5));

empSet.add(**new** Employee(7932, "Annie", 3000.0, "Female", 2.5));

empSet.add(**new** Employee(7691, "John", 4000.0, "Male", 3.5));

empSet.add(**new** Employee(7356, "Lisa", 9000.0, "Female", 6.5));

empSet.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

**return** empSet;

}

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

HashSet<Employee> setEmp = *getEmployeesSet*();

**for** ( Employee emp : setEmp ) {

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

}

}

}

Q) is a HashSet object, a thread-safe?

A) No, it is not a thread-safe object.

Q) Can we make HashSet object as thread-safe?

A) Yes. We can create a synchronized hash set object by calling synchronizedSet() method of Collections class.

ex:

Set<Employee> newSetEmp = Collections.synchronizedSet(setEmp);

Q) is a HashSet object, a mutable?

A) Yes. It is mutable object.

Q) can we make HashSet object as immutable?

A) Yes. We can create immutable hash set object, by calling unmodifiableSet() method of Collections class.

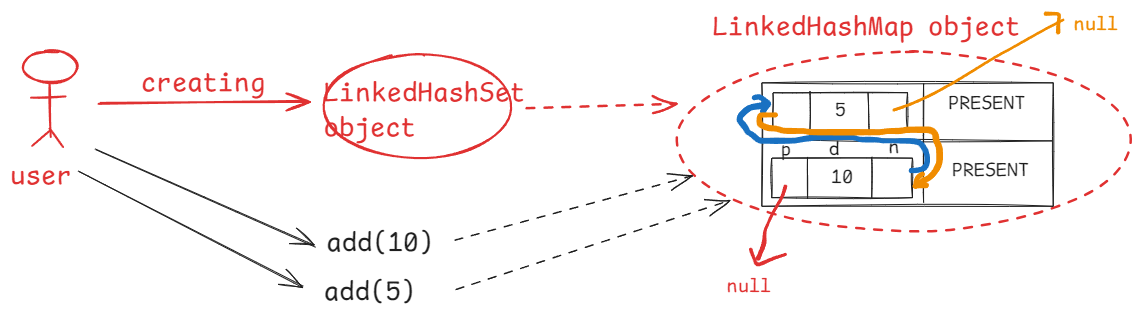
ex:

Set<Employee> newSetEmp = Collections.unmodifiableSet(setEmp);

LinkedHashSet class:

----------------------

* LinkedHashSet is subclass of HashSet.
* Use LinkedHashSet object, when you want to maintain the insertion order of the elements.
* LinkedHashSet internally uses LinkedHashMap object.



* The default initial capacity of internal LinkedHashMap is 16, and load factor 0.75.
* LinkedHashMap stores the keys as nodes, where each node has previous, data and next parts.
* Since, LinkedHashSet uses LinkedHashMap, the element is stored as a node and it has links to its previous element and also to the next element.
* For the first element added, its previous link has null and for the last element added, its next link has null.
* The additional methods of LinkedHashSet class given Java SE 21 are,

1.addFirst(E e): adds the element as a first element to this collection.

2.addLast(E e): adds the element as a last element to this collection.

3.getFirst(): returns the first element of this collection. Throws NoSuchElementException, if this collection is empty.

4. getLast(): returns the last element of this collection. Throws NoSuchElementException, if this collection is empty.

5. removeFirst(): remvoes and returns the first element of this collection. Throws NoSuchElementException, if this collection is empty.

6. removeLast(): removes and returns the last element of this collection. Throws NoSuchElementException, if this collection is empty.

7.reversed(): returns the reversed elements of this collection.

Author.java

**package** pack1;

**public** **class** Author {

**private** **int** authorId;

**private** String authorName;

**public** Author(**int** authorId, String authorName) {

**super**();

**this**.authorId = authorId;

**this**.authorName = authorName;

}

**public** **int** getAuthorId() {

**return** authorId;

}

**public** **void** setAuthorId(**int** authorId) {

**this**.authorId = authorId;

}

**public** String getAuthorName() {

**return** authorName;

}

**public** **void** setAuthorName(String authorName) {

**this**.authorName = authorName;

}

@Override

**public** String toString() {

**return** "Author [authorId=" + authorId + ", authorName=" + authorName + "]";

}

}

Book.java

**package** pack1;

**import** java.util.List;

**public** **class** Book {

**private** **int** bookId;

**private** String bookName;

**private** List<Author> lstAuthors;

**public** Book(**int** bookId, String bookName, List<Author> lstAuthors) {

**super**();

**this**.bookId = bookId;

**this**.bookName = bookName;

**this**.lstAuthors = lstAuthors;

}

**public** **int** getBookId() {

**return** bookId;

}

**public** **void** setBookId(**int** bookId) {

**this**.bookId = bookId;

}

**public** String getBookName() {

**return** bookName;

}

**public** **void** setBookName(String bookName) {

**this**.bookName = bookName;

}

**public** List<Author> getLstAuthors() {

**return** lstAuthors;

}

**public** **void** setLstAuthors(List<Author> lstAuthors) {

**this**.lstAuthors = lstAuthors;

}

@Override

**public** String toString() {

**return** "Book [bookId=" + bookId + ", bookName=" + bookName + ", lstAuthors=" + lstAuthors + "]";

}

}

Solution.java

**package** pack1;

**import** java.util.Arrays;

**import** java.util.Iterator;

**import** java.util.LinkedHashSet;

**public** **class** Solution {

**private** **static** LinkedHashSet<Book> getBooks() {

LinkedHashSet<Book> booksLHS = **new** LinkedHashSet<>();

booksLHS.add(**new** Book(3167, "Automic Habits", Arrays.*asList*(**new** Author(1, "James"), **new** Author(2, "Collins"))));

booksLHS.add(**new** Book(2765, "The Power", Arrays.*asList*(**new** Author(3, "Stephen"), **new** Author(4, "Mark"))));

booksLHS.add(**new** Book(4221, "Zero to One", Arrays.*asList*(**new** Author(1, "James"), **new** Author(5, "Simon"), **new** Author(8, "Gibson"))));

booksLHS.add(**new** Book(1754, "The Startup", Arrays.*asList*(**new** Author(6, "Peter"), **new** Author(2, "Collins"))));

booksLHS.add(**new** Book(1901, "Leaders Eat Last", Arrays.*asList*(**new** Author(4, "Mark"), **new** Author(7, "Jim"), **new** Author(6, "Peter"))));

**return** booksLHS;

}

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

LinkedHashSet<Book> booksLHS = *getBooks*();

Iterator<Book> it = booksLHS.iterator();

**while**(it.hasNext()) {

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

}

}

}

Q) is a LinkedHashSet object, a thread-safe?

A) No, it is not a thread-safe object.

Q) Can we make LinkedHashSet object as thread-safe?

A) Yes. We can create a synchronized linked hash set object by calling synchronizedSet() method of Collections class.

ex:

Set<Employee> newSetEmp = Collections.synchronizedSet(linkedHSEmp);

Q) is a LinkedHashSet object, a mutable?

A) Yes. It is mutable object.

Q) can we make LinkedHashSet object as immutable?

A) Yes. We can create immutable hash set object, by calling unmodifiableSet() method of Collections class.

ex:

Set<Employee> newSetEmp = Collections.unmodifiableSet(linkedHSEmp);

TreeSet class:

* TreeSet class implements NavigableSet interface and this NavigableSet interface extends SortedSet interface and this SortedSet interface extends Set interface.
* TreeSet class internally uses TreeMap object.
* The default initial capacity is 16 and the load factor is 0.75.
* TreeSet object maintains elements in natural sorting order by default. So the elements should be Comparable objects.
* If you wish to maitain the TreeSet elements in a custom sorting order then you can provide Comparator object while constructing TreeSet object.
* Methods of SortedSet interface are,
* 1.first()
* 2.last()
* 3.headSet(toElement): returnns a Set, whose elements are less than the toElement.
* 4.tailSet(fromElement): returns a Set, whose elements are greater than or equals to the fromElement.
* 5.subSet(fromElement, toElement): returns a Set, which includes fromElement, and excludes toElement.
* methods of NavigableSet interface are,

1.higher(element): returns the least element greater than the given element, or null if there is no such element.

2.lower(element): returns the greatest element less than the given eleement, or null if there is no such element.

3.ceiling(element): returns the least element greater than or equals to the given element, or null if there is no such element.

4.floor(element): returns the greatest element less than or equals to the given eleement, or null if there is no such element.

5.pollFirst(): retrieves and removes the first element, or null if this set is empty.

6.pollLast(): retrieves and removes the last element, of null if this set is empty.

7.descendingSet(): returns a reverse order of the elements of this set.

example1: with natural sorting order, using Comparable.

**package** pack1;

**import** java.util.Objects;

**import** java.util.TreeSet;

**class** Employee **implements** Comparable<Employee> {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(empno, ename, experience, gender, sal);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**return** empno == other.empno && Objects.*equals*(ename, other.ename)

&& Double.*doubleToLongBits*(experience) == Double.*doubleToLongBits*(other.experience)

&& Objects.*equals*(gender, other.gender)

&& Double.*doubleToLongBits*(sal) == Double.*doubleToLongBits*(other.sal);

}

@Override

**public** **int** compareTo(Employee o) {

**return** **this**.getEmpno() - o.getEmpno();

}

}

**public** **class** Solution {

**private** **static** TreeSet<Employee> getEmployeesSet() {

TreeSet<Employee> empSet = **new** TreeSet<>();

empSet.add(**new** Employee(7298, "Scott", 5000.0, "Male", 4.5));

empSet.add(**new** Employee(7178, "Allen", 7000.0, "Male", 5.5));

empSet.add(**new** Employee(7154, "Kathey", 6000.0, "Female", 4.5));

empSet.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

empSet.add(**new** Employee(7741, "Mary", 4000.0, "Female", 3.1));

empSet.add(**new** Employee(7865, "Daniel", 9000.0, "Male", 7.5));

empSet.add(**new** Employee(7539, "Jennie", 7000.0, "Female", 5.5));

empSet.add(**new** Employee(7932, "Annie", 3000.0, "Female", 2.5));

empSet.add(**new** Employee(7691, "John", 4000.0, "Male", 3.5));

empSet.add(**new** Employee(7356, "Lisa", 9000.0, "Female", 6.5));

empSet.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

**return** empSet;

}

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

TreeSet<Employee> treeSet = *getEmployeesSet*();

System.***out***.println("The order of elements in the TreeSet object");

**for** ( Employee emp : treeSet ) {

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

}

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

System.***out***.println("First Element : "+ treeSet.first());

System.***out***.println("Last element : "+ treeSet.last());

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

treeSet.pollFirst();

treeSet.pollLast();

System.***out***.println("First element & last element after removal : ");

System.***out***.println("First Element : "+ treeSet.first());

System.***out***.println("Last element : "+ treeSet.last());

}

}

example2: with custom sorting order, with Comparator

**package** pack1;

**import** java.util.Comparator;

**import** java.util.Objects;

**import** java.util.TreeSet;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** hashCode() {

**return** Objects.*hash*(empno, ename, experience, gender, sal);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**return** empno == other.empno && Objects.*equals*(ename, other.ename)

&& Double.*doubleToLongBits*(experience) == Double.*doubleToLongBits*(other.experience)

&& Objects.*equals*(gender, other.gender)

&& Double.*doubleToLongBits*(sal) == Double.*doubleToLongBits*(other.sal);

}

}

**class** NameComparator **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee o1, Employee o2) {

**return** o1.getEname().compareTo(o2.getEname());

}

}

**public** **class** Solution {

**private** **static** TreeSet<Employee> getEmployeesSet() {

TreeSet<Employee> empSet = **new** TreeSet<>(**new** NameComparator());

empSet.add(**new** Employee(7298, "Scott", 5000.0, "Male", 4.5));

empSet.add(**new** Employee(7178, "Allen", 7000.0, "Male", 5.5));

empSet.add(**new** Employee(7154, "Kathey", 6000.0, "Female", 4.5));

empSet.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

empSet.add(**new** Employee(7741, "Mary", 4000.0, "Female", 3.1));

empSet.add(**new** Employee(7865, "Daniel", 9000.0, "Male", 7.5));

empSet.add(**new** Employee(7539, "Jennie", 7000.0, "Female", 5.5));

empSet.add(**new** Employee(7932, "Annie", 3000.0, "Female", 2.5));

empSet.add(**new** Employee(7691, "John", 4000.0, "Male", 3.5));

empSet.add(**new** Employee(7356, "Lisa", 9000.0, "Female", 6.5));

empSet.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

**return** empSet;

}

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

TreeSet<Employee> treeSet = *getEmployeesSet*();

System.***out***.println("The order of elements in the TreeSet object");

**for** ( Employee emp : treeSet ) {

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

}

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

System.***out***.println("First Element : "+ treeSet.first());

System.***out***.println("Last element : "+ treeSet.last());

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

treeSet.pollFirst();

treeSet.pollLast();

System.***out***.println("First element & last element after removal : ");

System.***out***.println("First Element : "+ treeSet.first());

System.***out***.println("Last element : "+ treeSet.last());

}

}

Queue interface:

---------------

* Queue orders elements in First In First Out manner, but not necessarily.
* Queue maintains head and tail pointers.
* In FIFO queue, the elements are inserted from the tail of the queue and elements are removed from the head of the queue.
* To add the elements, methods are add(E e) and offer(E e)
* To remove the elements, methods are remove() and poll()
* To examine the elements, methods are element() and peek().

add(E e) : throws exception, if the element is not added.

offer(E e): returns false, if the element is not added

remove(): removes the head of the queue. If queue is empty then throws an exception.

poll(): removes the head of the queue. If queue is empty thren returns null.

element(): returns but does not remove the head of the queue. If queue is empty then throws an exception.

peek(): retruns but does not remove the head of the queue. If queue is empty then returns null.

PriorityQueue: It is a class, which does not maintain the elements in FIFO order.

* + It stores the elements in either natural order or the Comparator order.
  + The default capacity of the priority queue is 11.
  + If we add 12th element, the capacity will grow to 12. If we add 13th, the capacity will grow to 13 and so on. There is no limit. So it is an unbounded queue.
  + For ordering in natural order, the priority queue, relies on Comparable objects. So it does permit insertion of non-comparable elements.
  + We can not add null value to the priority queue.
  + if we add null, then we won’t get compile-time error. But we get NullPointerException at runtime.

//code on PriorityQueue class.

**package** pack1;

**import** java.util.PriorityQueue;

**class** Employee **implements** Comparable<Employee> {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

@Override

**public** **int** compareTo(Employee o) {

**return** **this**.getEmpno() - o.getEmpno();

}

}

**public** **class** Solution {

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

PriorityQueue<Employee> pque = **new** PriorityQueue<>();

pque.add(**new** Employee(7298, "Scott", 5000.0, "Male", 4.5));

pque.add(**new** Employee(7178, "Allen", 7000.0, "Male", 5.5));

pque.add(**new** Employee(7154, "Kathey", 6000.0, "Female", 4.5));

pque.add(**new** Employee(7233, "Clark", 5000.0, "Male", 4.5));

pque.add(**new** Employee(7741, "Mary", 4000.0, "Female", 3.1));

pque.add(**new** Employee(7865, "Daniel", 9000.0, "Male", 7.5));

pque.add(**new** Employee(7539, "Jennie", 7000.0, "Female", 5.5));

pque.add(**new** Employee(7932, "Annie", 3000.0, "Female", 2.5));

pque.add(**new** Employee(7691, "John", 4000.0, "Male", 3.5));

pque.add(**new** Employee(7356, "Lisa", 9000.0, "Female", 6.5));

Employee e=**null**;

**while**( (e = pque.poll()) != **null**) {

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

}

}

}

BlockingQueue:

* It is queue, which additionally supports blocking operations.
* A blocking queue will have a fixed capacity.
* If a thread wants to add an element, but the queue is full, then the thread has to wait until space becomes available.
* If a thread wants to remove an element, but the queue is empty, then the thread has to wait until element becomes available.
* The blocking operations are put() and take().
* When put() is called, if the queue is full, put() will wait for the space.
* when take() is called, if the queue is empty, take() will wait for the element.
* The implementation classes are, ArrayBlockingQueue and LinkedBlockingQueue.
* For creating producer-consumer applications, producer adds items to the queue and consumer remvoes items from the queue.
* If the queue is full, producer waits until, space becomes available. If the queue is empty, consumer waits until, item becomes available.
* For example, In a Job execution application, one thread is adding the job to the queue and the other thread is removing the job from the queue and executing it.
* If the queue if full, the first thread waits to add the job, until space becomes available. If the queue is empty, the other threads waits until a job is added.

//example

**public** **class** Solution {

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

ArrayBlockingQueue<Integer> abq = **new** ArrayBlockingQueue<>(5);

abq.put(10);;

abq.put(20);;

abq.put(30);;

abq.put(40);;

abq.put(50);

System.***out***.println("The capacity is 5, and 5 elements are added");

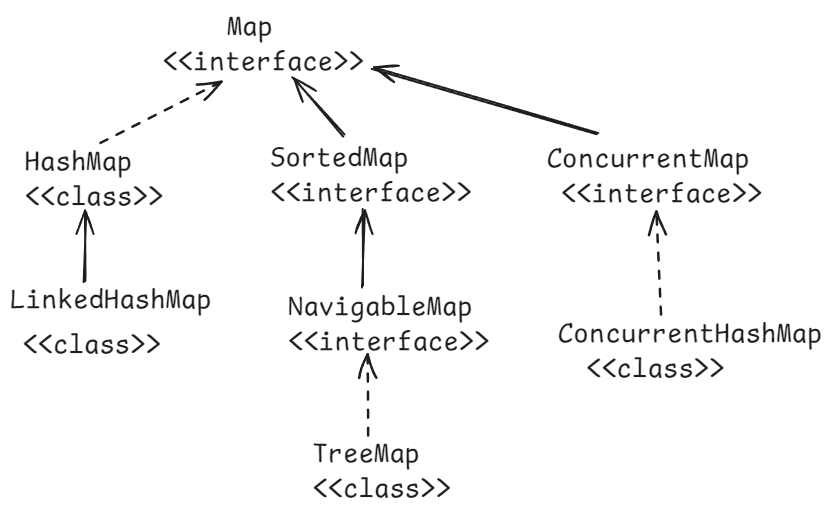
System.***out***.println("trying to add 6th element, waiting for the space");

abq.put(60);

}

}

Map interface:



HashMap class:

* When you want to store the elements as a key and value pairs then choose Map object.
* In a Map object, key can’t be duplicated, but value can be duplicated.
* we can insert one null key, and multiple null values into a Map object.
* When the HashMap object is created, the object is created with initial capacity 16 and a load factor 0.75.
* Each key-value pair is called an entry to the map object.
* The HashMap object doesn’t maintain insertion order of the entries into the map object.
* Some import methods of HashMap class are,
  + 1. put(key,value): inserts the key-value to the map and returns the value.
    2. get(key): retruns the value of the specified key. Returns null if the key doesn’t exist.
    3. remove(key): removes the mapping from this map object for the key, and returns the value. If the map is empty, then returns null.
    4. replace(key,value): replaces the value for a key in this map, if the key exists. Otherwise returns null.
    5. containsKey(key): returns true, if this map object contains the specified key. Otherwise, returns false.
    6. containsValue(value): returns true, if this map object contains the specified value. Otherwise, returns false.
    7. keySet() : returns keys of this map object into a Set. Returns null if this map object is empty.
    8. values(): returns values of this map object into a set. Returns null if this map object is empty.
    9. entrySet(): returns the map entries as Map.Entry objects into a set. Returns null, if the map object is empty.

//example1

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** Solution {

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

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

hashMap.put(101, "Scott");

hashMap.put(105, "Lisa");

hashMap.put(103, "Allen");

hashMap.put(102, "Mary");

hashMap.put(104, "David");

hashMap.put(**null**, **null**); //allowed

hashMap.put(**null**, "Clark"); //allowed, but the value "Clark" replaces

// the value null

hashMap.put(106, **null**); //allowed

hashMap.put(109, **null**); //allowed

//call keySet()

Set<Integer> keys = hashMap.keySet();

System.***out***.println("The keys of this map object");

**for**(Integer i : keys ) {

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

}

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

System.***out***.println("=".repeat(40));

//call values()

Collection<String> values = hashMap.values();

System.***out***.println("The values of this map object");

**for**(String s : values) {

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

}

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

System.***out***.println("=".repeat(40));

//call entrySet()

Set<Map.Entry<Integer, String>> entries = hashMap.entrySet();

System.***out***.println("The key-value mappings of this map object");

**for**(Map.Entry<Integer, String> entry : entries) {

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

}

}

}

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**class** Employee {

**private** **int** empno;

**private** String ename;

**private** **double** sal;

**private** String gender;

**private** **double** experience;

**public** Employee(**int** empno, String ename, **double** sal, String gender, **double** experience) {

**super**();

**this**.empno = empno;

**this**.ename = ename;

**this**.sal = sal;

**this**.gender = gender;

**this**.experience = experience;

}

**public** **int** getEmpno() {

**return** empno;

}

**public** **void** setEmpno(**int** empno) {

**this**.empno = empno;

}

**public** String getEname() {

**return** ename;

}

**public** **void** setEname(String ename) {

**this**.ename = ename;

}

**public** **double** getSal() {

**return** sal;

}

**public** **void** setSal(**double** sal) {

**this**.sal = sal;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** **double** getExperience() {

**return** experience;

}

**public** **void** setExperience(**double** experience) {

**this**.experience = experience;

}

@Override

**public** String toString() {

**return** "Employee [empno=" + empno + ", ename=" + ename + ", sal=" + sal + ", gender=" + gender + ", experience="

+ experience + "]";

}

}

**public** **class** Solution {

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

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

Employee e1 = **new** Employee(7298, "Scott", 5000.0, "Male", 4.5);

Employee e2 = **new** Employee(7741, "Mary", 4000.0, "Female", 3.1);

Employee e3 = **new** Employee(7691, "John", 4000.0, "Male", 3.5);

Employee e4 = **new** Employee(7233, "Clark", 5000.0, "Male", 4.5);

hashMap.put(e1.getEmpno(), e1);

hashMap.put(e2.getEmpno(), e2);

hashMap.put(e3.getEmpno(), e3);

hashMap.put(e4.getEmpno(), e4);

Employee e = hashMap.get(7741);

System.***out***.println("The employee of number 7741 :");

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

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

Set<Map.Entry<Integer,Employee>> entries = hashMap.entrySet();

**for**(Map.Entry<Integer, Employee> entry : entries ) {

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

}

}

}

Q) What is the difference between HashMap and Hashtable classes?

A) 1. HashMap is a collection framework class.

Hashtable is a legacy class, given before

collection framework.

Note: Hashtable is given in JDK1.0, but

collection framework is introducted JDK1.2

2. HashMap allows null key and null values.

Hashtable doesn’t allow null key or null value.

If you add, then at runtime, NullPointerException

will be thrown.

3. HashMap is not thread-safe object. It means,

multiple threads can work on the HashMap object

at a time.

Hashtable is a thread-safe object. It means,

only one thread can work on the Hashtable object

at a time.

4. HashMap object’s performance is fast.

Hashtable object’s performance is slow.

5. Elements of HashMap object can be sorted, by

converting HashMap object to a TreeMap object.

Elements of Hashtable object can’t be sorted.

Q) is a HashMap object, a thread-safe?

A) No, it is not a thread-safe object.

Q) Can we make HashMap object as thread-safe?

A) Yes. We can create a synchronized HashMap object by calling synchronizedMap() method of Collections class.

ex:

Map<Integer,Employee> newMapEmp = Collections.synchronizedMap(mapEmp);

Q) is a HashMap object, a mutable?

A) Yes. It is mutable object.

Q) can we make HashMap object as immutable?

A) Yes. We can create immutable hash map object, by calling unmodifiableMap() method of Collections class.

ex:

Map<Integer,Employee> newMapEmp = Collections.unmodifiableMap(mapEmp);

Q) What is Properties class?

A) . Properties is a legacy class extends Hashtable

class.

. Properties class can store key-value pairs of

String type only.

. Properties class can also load the key-value pairs

from a file, by using its load() method.

Q) what is the difference between Enumeration and Iterator?

A) 1. Enumeration is a legacy interface.

Iterator is a collection framework interface.

2. Enumeration interface contains methods

hasMoreElements()

nextElement()

Iterator interface contains methods

hasNext()

next()

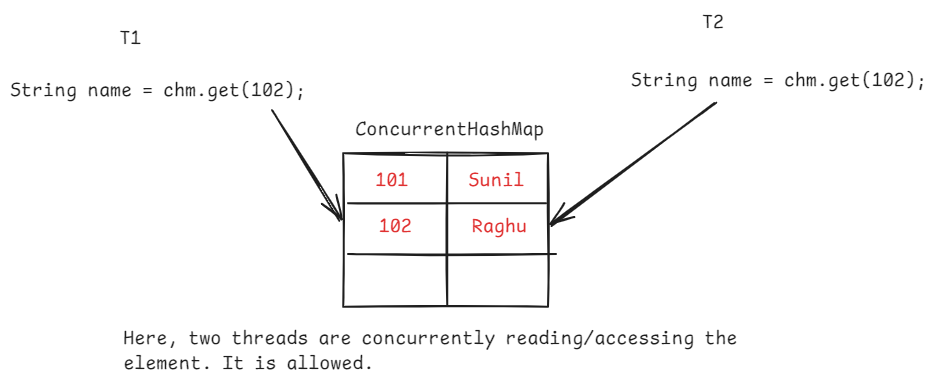
remove()

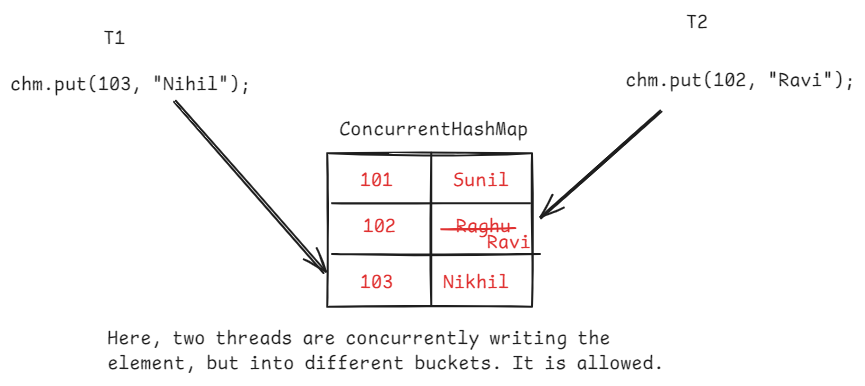
Q) can we store a custom class objects as a key in HashMap object?

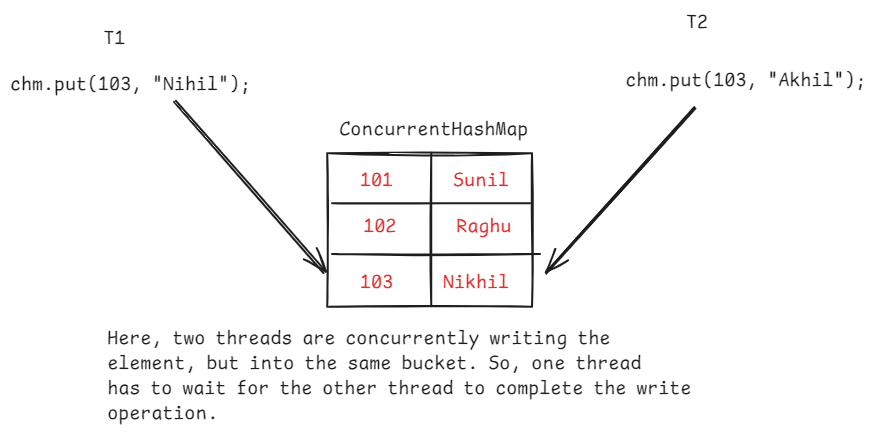
A) Yes. But it is recommended to override hashCode() and equals() method in that class.

ConcurrentHashMap :

* ConcurrentHashMap class implements ConcurrentMap interface and ConcurrentMap interface extends Map interface.
* ConcurrentHashMap class is a thread-safe variant of HashMap class.
* ConcurrentHashMap allows multiple threads at a time to access its elements.
* ConcurrentHashMap provides better performance in multi-threading applications, because of bucket-level locking.
* ConcurrentHashMap also has default initial capacity 16 with load factor 0.75.







Q) What is difference between ConcurrentHashMap and HashMap?

A) 1. ConcurrentHashMap is a thread-safe object. It

means, multiple threads can concurrently

read/write elements without causing

inconsistency.

HashMap is a not a thread-safe object. It

means, if multiple threds are concurrently

writing the elements, then inconsistency

occurs.

2. ConcurrentHashMap does not allow null key or

value.

HashMap allows one null key and multiple null

values.

3. ConcurrentHashMap provides better performance

in multithreaded applications.

HashMap provides bettern performance in single

threaded applications.

4. ConcurrentHashMap iteration is fail-safe.

HashMap iteration is fail-fast.

Fail-fast and Fail-safe iterator:

* Fail-fast and Fail-safe are the terms, they refer how iterators behave when the underlying collection is structurally modified while iterating over it.
* Fail-fast iterator means, while an iterator is iterating on a collection, if that collection is modified by adding/removing element, then the iterator throws ConcurrentModificationException immediately.
* For example, While iterating over ArrayList object, if a new element is added or an element is removed from the ArrayList object, then immediately the iterator throws ConcurrentModificationException.
* Fail-safe iterators do not throw exception, when the collection is modified during the iteration.
* Fail-safe iterators work on the copy of the original collection object, but not on the original object. So, if any changes are made to the original collection object, does not reflect in the copy object.
* The iterators created on ArrayList, LinkedList, HashSet, LinkedHashSet, HashMap, TreeSet, LinkedHashMap, TreeMap, Queue, etc.. are fail-fast iterators.
* The iterators created on ConcurrentHashMap, CopyOnWriteArrayList, etc.. are fail-safe iterators.

**import** java.util.Iterator;

**import** java.util.concurrent.ConcurrentHashMap;

**public** **class** Solution {

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

/\*

List<String> lst = new ArrayList<String>();

lst.add("A");

lst.add("B");

lst.add("C");

Iterator<String> it = lst.iterator();

while(it.hasNext()) {

System.out.println(it.next());

lst.remove("C");

}

\*/

ConcurrentHashMap<String, String> map = **new** ConcurrentHashMap<String, String>();

map.put("A", "Apple");

map.put("B", "Banana");

map.put("C", "Coffee");

//iterating over the keys

Iterator<String> it = map.keySet().iterator();

**while**(it.hasNext()) {

String key = it.next();

System.***out***.println("key : " + key);

map.put("D", "Doll");

}

}

}