**Generic & collections Framework**

The main use of generics is to be able to write reusable (and type-safe) data structures and algorithms. The Java library has a collections framework that makes extensive use of generics and provides a set of containers and algorithms.

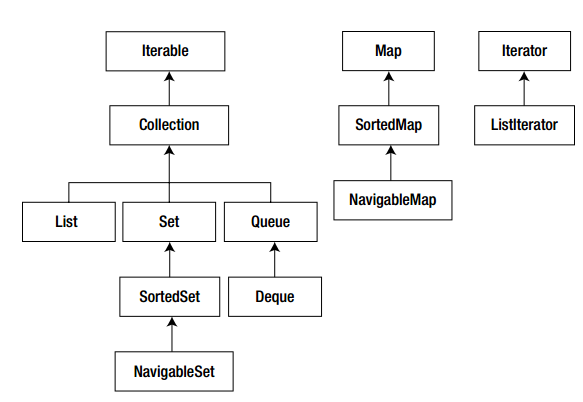
you can just use the data structures readily available in the Java library rather than implementing them yourself.

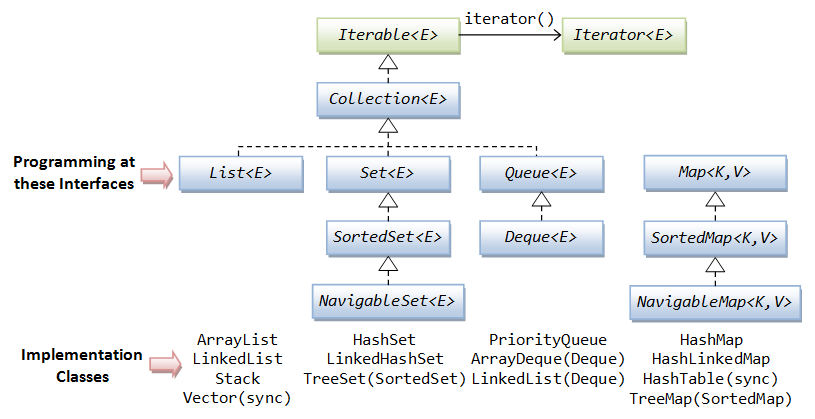
**Using Generic: common DAO; Generics Services 🡺 Examples**

Basic Components of the Collections Framework

* **Abstract classes and interfaces**
* **Concrete classes**
* **Algorithms:** The java.util.Collections implements commonly require functionality like sorting, searching, etc.

java.util **Abstract Classes and Interfaces**





Important method in the collection Interface

**Method Short description**  
boolean add(Element elem) Adds elem into the underlying container.  
void clear() Removes all elements from the container.  
boolean isEmpty() Checks whether the container has any elements or not.  
Iterator<Element> iterator() Returns an Iterator<Element> object for iterating over the container.  
boolean remove(Object obj) Removes the element if obj is present in the container.  
int size() Returns the number of elements in the container.  
Object[] toArray() Returns an array that has all elements in the container.

Concrete Classes

|  |  |
| --- | --- |
| **Concrete Class** | Note |
| ArrayList | Fast to search, but slow to insert or delete. Allows duplicates. |
| LinkedList | Fast to insert or delete elements, but slow for searching elements. (LIFO) or queue (FIFO) data structure. Allows duplicates. |
| HashSet | Fast for searching and retrieving elements, not maintain any order, Not allow duplicates |
| TreeSet | Like HashSet but store element in a sorted order |
| HashMap | Searching or inserting is very fast; not maintain any order |
| TreeMap | store element in a sorted order, position is decided by the sorting order |
| PriorityQueue | A PriorityQueue is for retrieving elements based on priority. |
|  |  |

The Iterator **Interface**

three methods: **hasNext() , next() ,** and **remove()**

List Classes: Two concrete classes including **ArrayList** and **LinkedList**.

**ArrayList Class**

//// This program shows the usage of Iterator

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** Abc {

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

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

//🡺 should use List<Integer> nums = new ArrayList<Integer>();

**for** (**int** i = 1; i < 10; i++)

nums.add(i);

System.***out***.println("Original list " + nums);

Iterator<Integer> numsIter = nums.iterator();

**while** (numsIter.hasNext()) {

numsIter.next();

numsIter.remove();

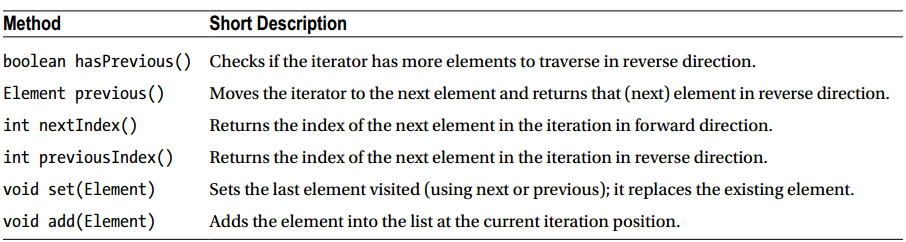
}

System.***out***.println("List after removing all elements" + nums);

}

}

The ListIterator Interface:



**The LinkedList Class:** insertion and deletion is very fast

**import** java.util.\*;

**public** **class** Abc {

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

String palStr = "abcba";

List<Character> palindrome = **new** **LinkedList**<Character>();

**for** (**char** ch : palStr.toCharArray())

palindrome.add(ch);

System.***out***.println("Input string is: " + palStr);

**ListIterator**<Character> iterator = palindrome.listIterator();

**ListIterator**<Character> revIterator = palindrome.listIterator(palindrome.size());

**boolean** result = **true**;

**while** (revIterator.**hasPrevious**()) {

**if** (iterator.next() != revIterator.**previous**()) {

result = **false**;

**break**;

}

}

**if** (result)

System.***out***.print("Input string is a palindrome");

**else**

System.***out***.print("Input string is not a palindrome");

}

}

The difference between Iterator & ListIterator

1) **Iterator** is used for traversing **List** and **Set** both. We can use **ListIterator** to traverse **List only**, we cannot traverse Set using **ListIterator**.

2) We can traverse in only forward direction using **Iterator**. Using **ListIterator**, we can traverse a List in **both the directions (forward and Backward)**.

The Set Interface

**The HashSet Class**  
Given a sentence, how can you remove repeated words in that sentence? Set does not allow duplicates, and HashSet can be used for quick insertion and search. So you can use a HashSet for solving this problem

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

String tongueTwister = "I feel, a feel, a funny feel, a funny feel I feel, "

+ "if you feel the feel I feel, I feel the feel you feel";

Set<String> words = **new** HashSet<>();

// split the sentence into words and try putting them in the set

**for**(String word : tongueTwister.split("\\W+"))

words.add(word);

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

}

**[the, a, I, feel, if, funny, you]**

**The TreeSet Class**

**Same at HashSet but all item in ordered list**

If we replace the HashSet by TreeSet in the above code as following:

Set<String> words = new TreeSet<>();

**[I, a, feel, funny, if, the, you]**

**MAP**

Have two important concrete classes of Map that we’ll cover: **HashMap** and **TreeMap**.HashMap help us retrieve quickly using hashtable but don’t store the order. TreeMap using red-black tree, so it store the order but slower than HashMap.

**The HashMap Class**

Given an input string, the spell checker looks for words that are usually misspelled; if there is a match, it prints the correct spelling. So, the spell checker should maintain a list of frequently misspelled words and their correct spellings. How can you implement this?

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

misspeltWords.put("calender", "calendar");

misspeltWords.put("tomatos", "tomatoes");

misspeltWords.put("existance", "existence");

misspeltWords.put("aquaintance", "acquaintance");

String sentence = "Buy a calender for the year 2013";

System.***out***.println("The given sentence is: " + sentence);

**for** (String word : sentence.split("\\W+")) {

**if** (misspeltWords.containsKey(word)) {

System.***out***.println("The correct spelling for " + word + " is: " + misspeltWords.get(word));

}

}

* You can get all the keys in the HashMap using the keySet() method.

Set<String> keys = misspeltWords.keySet();

Result of the keys: [calender, existance, aquaintance, tomatos]

* Similarly, you can use valueSet() method to get the values

**Overriding the hashCode() Method**

It is important for using the classes with containers (particularly, **HashMap** and **HashSet**)

if you’re using an object in containers like HashSet or HashMap, make sure you override the **hashCode**() and **equals**() methods correctly. if you don’t, you’ll get **nasty surprises (bugs)** while using these containers!

Example:

**import** java.util.Objects;

**public** **class** Circle {

**private** **int** xPos, yPos, radius;

**public** Circle(**int** x, **int** y, **int** r) {

xPos = x;

yPos = y;

radius = r;

}

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

**if** (arg == **null**)

**return** **false**;

**if** (**this** == arg)

**return** **true**;

**if** (arg **instanceof** Circle) {

Circle that = (Circle) arg;

**if** ((**this**.xPos == that.xPos) && (**this**.yPos == that.yPos) && (**this**.radius == that.radius)) {

**return** **true**;

}

}

**return** **false**;

}

}

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

Set<Circle> circleList = **new** HashSet<Circle>();

circleList.add(**new** Circle(10, 20, 5));

System.***out***.println(circleList.contains(**new** Circle(10, 20, 5)));

}

**It prints false (not true)! Why?**

Add hashCode method in Circle class will solve the problem ☺

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

**return** Objects.*hash*(**this**.xPos, **this**.yPos, **this**.radius);

}

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

The NavigableMap Interface

you can **navigate** the Map **easily**. You can get the nearest value matching the given key, all values less than or greater than the given key, etc

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

NavigableMap<Integer, String> examScores = **new** TreeMap<Integer, String>();

examScores.put(90, "Sophia");

examScores.put(20, "Isabella");

examScores.put(10, "Emma");

examScores.put(50, "Olivea");

System.***out***.println("The data in the map is: " + examScores);

System.***out***.println("The data descending order is: " + examScores.descendingMap());

System.***out***.println("The data in the map is: " + examScores);

System.***out***.println("Dwho passed the exam: " + examScores.tailMap(40));

System.***out***.println("The lowest mark is: " + examScores.firstEntry());

}

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

The Queue Interface

A Queue follows FIFO mechanism; you can create a LinkedList object and refer it through a Queue reference

**EX**:

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

Queue<String> loginSequence = **new** LinkedList<String>();

loginSequence.add("Harrison");

loginSequence.add("McCartney");

loginSequence.add("Starr");

loginSequence.add("Lennon");

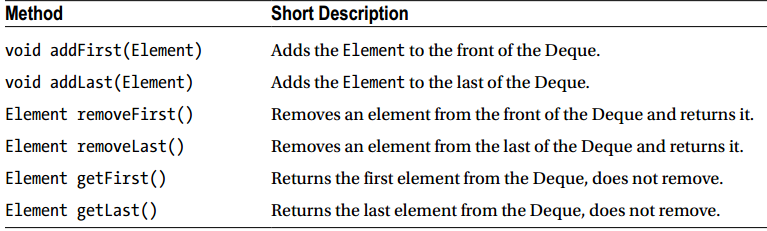
System.***out***.println("The login sequence is: " + loginSequence);

**while** (!loginSequence.isEmpty())

System.***out***.println("Removing " + loginSequence.remove());

}

The Deque Interface



There are three **concrete** implementations of the Deque interface: **LinkedList**, **ArrayDeque**, and **LinkedBlockingDeque**. Let’s use ArrayDeque to understand the features of the Deque interface.

**private** Deque<String> splQ = **new** ArrayDeque<>();

**void** addInQueue(String customer) {

splQ.addLast(customer);

}

**void** removeFront() {

splQ.removeFirst();

}

**void** removeBack() {

splQ.removeLast();

}

**void** printQueue() {

System.***out***.println("Special queue contains: " + splQ);

}

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

Comparable and Comparator Interfaces

To compare similar objects. The Comparable class has only one method compareTo() , which is declared as follows:

int compareTo(Element that)

Example: you have a list of student as the following info:

ID Name CGPA (for 4.0)

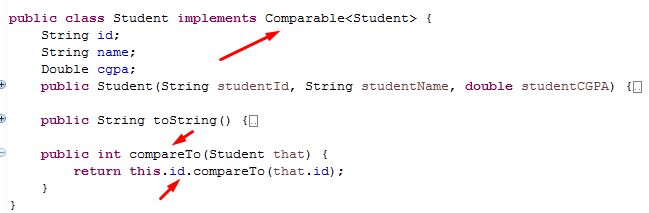
cs011 Lennon 3.1

cs021 McCartney 3.4

cs012 Harrison 2.7

cs022 Starr 3.7

If we want to sort by Student-ID or Name or CGPA with **natural comparison order, We** implement the **Comparable<Student>** interface. Because when we call sort(), it will call **compareTo**() method to compare objects based on what field we define in it.



If you need to implement **two or more alternative ways** to compare two similar objects, then you may implement the Comparator class

**public** **class** CGPAComparator **implements** Comparator<Student> {

**public** **int** compare(Student s1, Student s2) {

**if** (s1.id.compareTo(s2.id) == 0)

**return** (s1.cgpa.compareTo(s2.cgpa));

**return** (s1.id.compareTo(s2.id));

}

}

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

Student[] students = { **new** Student("cs011", "Lennon", 3.1), **new** Student("cs021", "McCartney", 3.4),

**new** Student("cs012", "Harrison ", 2.7), **new** Student("cs011", "aaST", 1.7) };

System.***out***.println("Before sorting by student ID");

System.***out***.println("Student-ID \t Name \t CGPA (for 4.0) ");

System.***out***.println(Arrays.*toString*(students));

**Arrays.*sort*(students, new CGPAComparator());**

System.***out***.println("After sorting by student ID");

System.***out***.println("Student-ID \t Name \t CGPA (for 4.0) ");

System.***out***.println(Arrays.*toString*(students));

}

Java 9

<https://www.oracle.com/java/java9-screencasts.html>