Generics & Collections

Agenda:

Java Generics

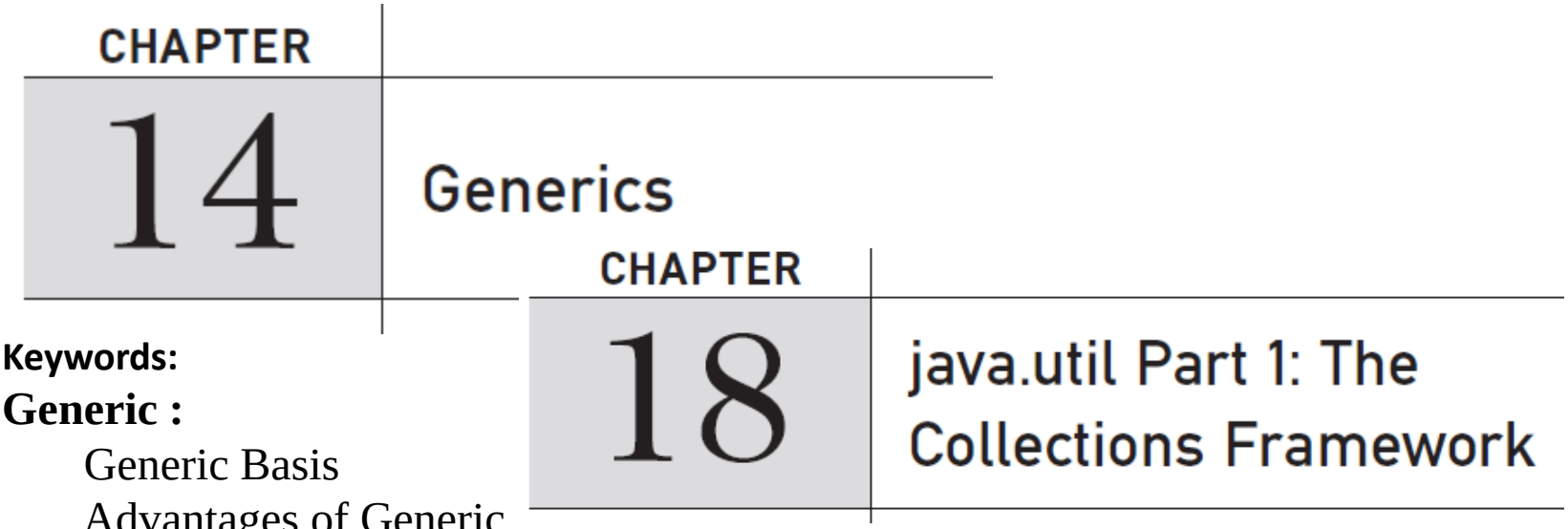
Java Collection Framework

Md. Mamun Hossain
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

Reference Text : Chap. -14,18

Generics & Collections



Keywords:

Generic :

- Generic Basis
- Advantages of Generic
- Generic Classes & Methods
- Type Safety :
  - Problem without generics
  - Solution using Generics

Collection: interfaces , Classes & Iterator

- interfaces(Set, List, Queue, Deque)
- classes(ArrayList, LikkedList, HashSet, HashMap etc.)

Java Collection

Generics in Java is similar to templates (STL) in C++.

The Parameterized Type

Generic

allow type (Integer, String, ... etc. and user defined types) ***to be a parameter to*** *methods, classes and interfaces.*

Generics in Java: Basis

Introduced by **JDK 5**, generics changed Java in two important ways.

- **First,** it added a new syntactical element to the language.
- **Second,** it caused changes to many of the classes and methods in the core API.

Generics in Java: Basis

- **Generics in Java is similar to templates (STL) in C++.**
- ***The idea is to*** *allow type (Integer, String, ... etc. and user defined types)* ***to be a parameter to*** *methods, classes and interfaces.*
- For example, ***classes*** like HashSet, ArrayList, HashMap, etc. use generics very well. We can use them for any type.

NB:
In parameter, we can not use primitives type like int, float, char or double

Generics in Java: Basis

- 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**, or **Thread**.

- 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 Classes

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()

Note: In Parameter type we can not use primitives like
      'int','char' or 'double'.
```

```
Stack<Integer> st = new Stack<Integer>();

Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type

TwoGen<Integer, String> tgObj =
  new TwoGen<Integer, String>(88, "Generics");

TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

General Form of Generics Classes

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

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

Here is the full syntax for declaring a reference to a generic class and instance creation:

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

```
Stack<Integer> st = new Stack<Integer>();

TwoGen<Integer, String> tgObj =
  new TwoGen<Integer, String>(88, "Generics");

TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

Exactly, What Are Generics?

At its core, the term ***generics*** means ***parameterized types***. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

Using generics, it is possible to create a single class, for example, that automatically works with different types of data.

A class, interface, or method that operates on a parameterized type is called *generic*, as in *generic class* or *generic method*.

Before, Java support generalized classes, interfaces, and methods by operating through references of type **Object**. But Type safety issue was still unsolved. *Generics added the type safety that was lacking*.

NOTE A Warning to C++ Programmers: Although generics are similar to templates in C++, they are not the same. There are some fundamental differences between the two approaches to generic types.

Generics Classes

```
Stack<Integer> st = new Stack<Integer>();
```

Perhaps the one feature of Java that has been most significantly affected by generics is the *Collections Framework*. The Collections Framework (Chapter 18) is part of the Java API.

A *collection* is a group of objects.

The Collections Framework defines several classes, such as lists and maps, that manage collections. The collection classes have always been able to work with any type of object.

The benefit that generics added is that the collection classes can now be used with complete type safety.

Generics Class Example

```
// We use < > to specify Parameter type
class Test<T>
{
    // An object of type T is declared
    T obj;
    Test(T obj) {  this.obj = obj;  }  // constructor
    public T getObject()  { return this.obj; }
}

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

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

Output:

```
15
GeeksForGeeks
```

Generics Class Example

```java
// A Simple Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U

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

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

We can also pass multiple Type parameters in Generic classes.

```java
// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("GfG", 15);

        obj.print();
    }
}
```

Output:

```
GfG
15
```

Generics Function

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
// A Simple Java program to show working of user defined
// Generic functions

class Test
{
    // A Generic method example
    static <T> void genericDisplay (T element)
    {
        System.out.println(element.getClass().getName() +
                           " = " + element);
    }

    // Driver method
    public static void main(String[] args)
    {
         // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("GeeksForGeeks");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

```
java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0
```

Generics: Advantages

- **Code Reuse:** We can write a method/class/interface once and use for any type we want.
- **Type Safety :** Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time).

  -Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.
- **Individual Type Casting is not needed:** If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.
- **Implementing generic algorithms:** By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

Generics: Advantages

There are mainly 3 advantages of generics. They are as follows:

1) **Type-safety :** We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

After Generics, we don't need to typecast the object

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

3) 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.

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

Type Safety: Problem without Generics

```java
// A Simple Java program to demonstrate that NOT using
// generics can cause run time exceptions
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creatinga an ArrayList without any type specified
        ArrayList al = new ArrayList();

        al.add("Sachin");
        al.add("Rahul");
        al.add(10); // Compiler allows this

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);

        // Causes Runtime Exception
        String s3 = (String)al.get(2);
    }
}
```

Output :

```
Exception in thread "main" java.lang.ClassCastExce
   java.lang.Integer cannot be cast to java.lang.S
        at Test.main(Test.java:19)
```

How generics solve this problem?

```java
// Using generics converts run time exceptions into
// compile time exception.
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Now Compiler doesn't allow this
        al.add(10);

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);
        String s3 = (String)al.get(2);
    }
}
```

How generics solve this problem? At the time of defining ArrayList, we can specify that this list can take only String objects.

Output:

```
15: error: no suitable method found for add(int)
        al.add(10);
          ^
```

Type Safety : Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time).

Generic: No individual Type casting

Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

```
// We don't need to typecast individual members of ArrayList
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Typecasting is not needed
        String s1 = al.get(0);
        String s2 = al.get(1);
    }
}
```

Java Collection

Iterator : ArrayList, LikkedList, HashSet, HashMap etc.

Java Collection Framework

interfaces (Iterator ← Collection: ←(*Set*, *List*, *Queue*, *Deque)*)
and
classes (*ArrayList*, *Vector*, *LinkedList*, *PriorityQueue*, *HashSet*, *LinkedHashSet*, *TreeSet*).

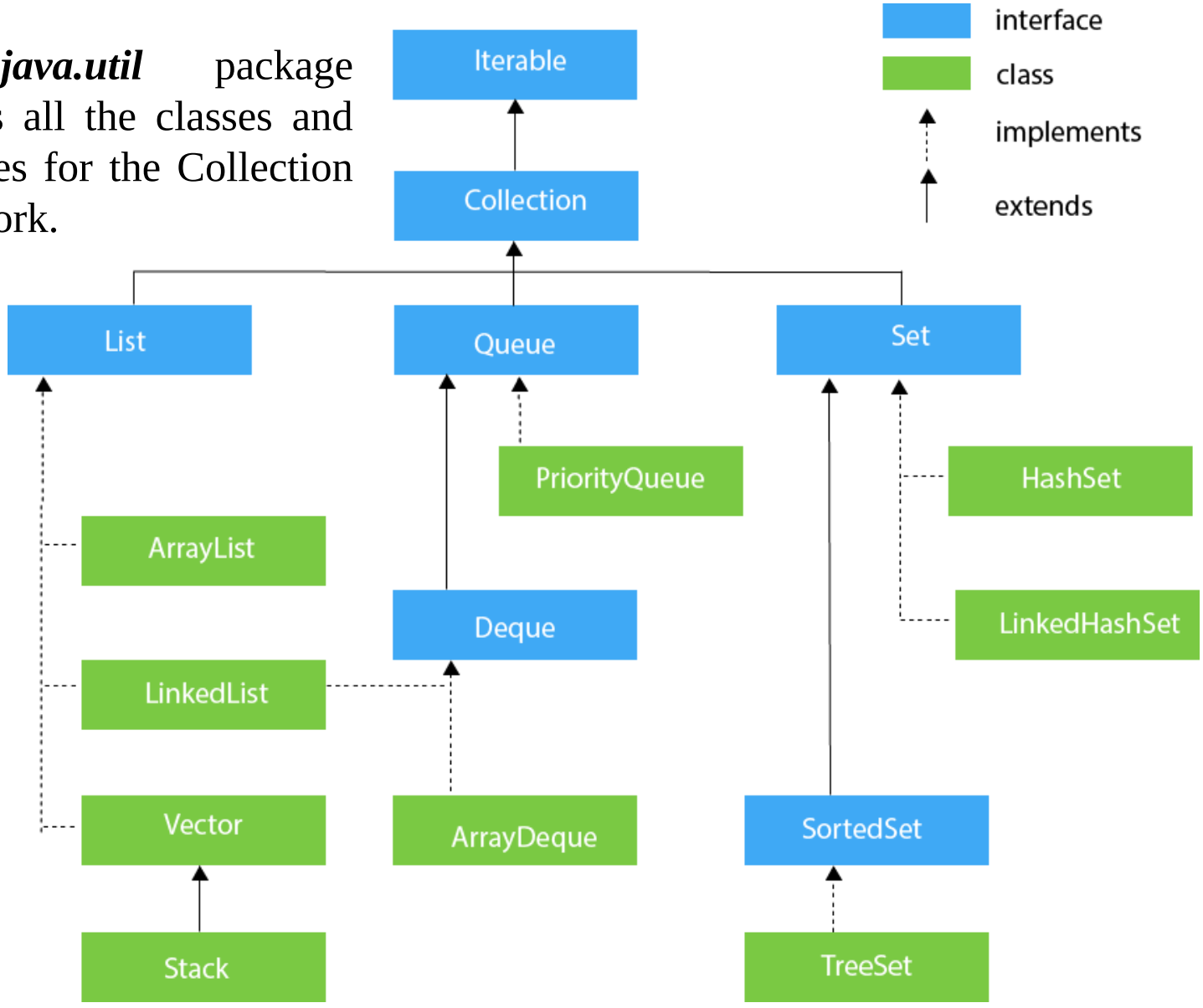
```
import java.util.*;
//import java.util.Iterator;
//import java.util.Collections;
//import java.util.List;
//import java.util.LnkedList;
//import java.util.ArrayList;
```

Java Collection

- Java Collection means a ***single unit of objects***.
- 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 framework provides many
  - **interfaces** (Set, List, Queue, Deque) and
  - **classes** (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

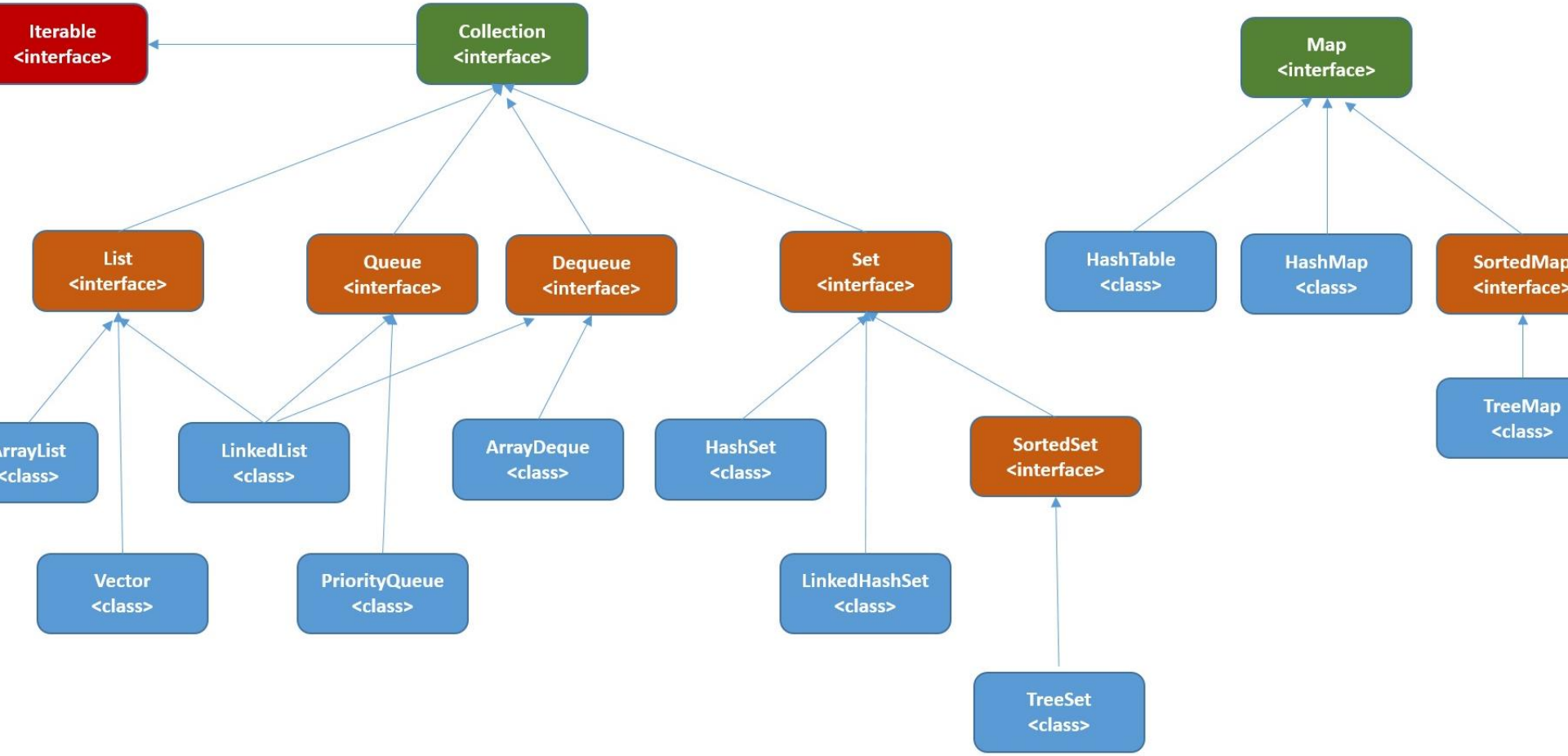
Collection Framework hierarchy

The ***java.util*** package contains all the classes and interfaces for the Collection framework.



Collection Framework hierarchy

The java.util package contains all the classes and interfaces for the Collection framework.



Collection Interfaces summary

The new (JDK1.2) general-purpose implementations of the collection interfaces are summarized in the table below:

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Complete list of Interfaces and Classes:

Set, List, Map, SortedSet, SortedMap, HashSet, TreeSet, ArrayList, LinkedList, Vector, HashMap, TreeMap, Hashtable, Collections, Arrays, AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList, AbstractMap, Iterator, ListIterator, Comparable, Comparator, UnsupportedOperationException, ConcurrentModificationException

Iterator

forward direction only

- Iterator interface provides the facility of iterating the elements in a of collections ***only in a forward direction.***

Java Collection : Iterator Interface

- **Iterator interface**

  Iterator interface provides the facility of iterating the elements in a ***forward direction only***.

- **Methods of Iterator interface**

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterator interface : Methods

```
package coll;
import java.util.Iterator;
import java.util.ArrayList;
public class Itr{
    public static void main(String[] args) {
        ArrayList <String> al=new ArrayList<String>();
            al.add("Sakib");
            al.add("Akib");
            al.add("Galib");
            al.add("Daya");
          System.out.println(al);
          Iterator it=al.iterator();
          while(it.hasNext())
              System.out.println(it.next());
        it.remove();
        System.out.println(al);
    }
}
```

Output:

```
[Sakib, Akib, Galib, Daya]
Sakib
Akib
Galib
Daya
[Sakib, Akib, Galib]
```

Java Collection : Iterator

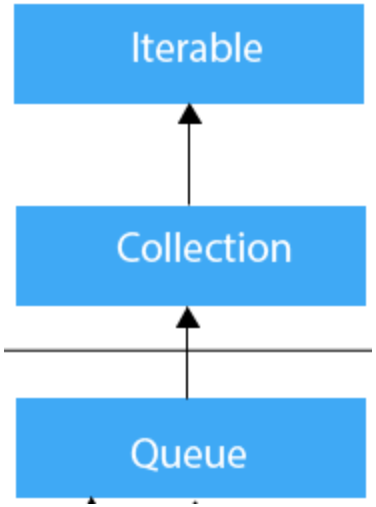
- **Iterator interface**

  The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

```
Iterator<T> iterator()
```

It returns the iterator over the elements of type T.



Example:

//Creating an arraylist *list*

*ArrayList<String> list=**new** ArrayList<String>();*

//Creating an iterator *it*

Iterator itr=list.iterator();

Java Collection : Iterator

Java Iterator

- An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.
- To use an Iterator, you must import it from the ***java.util*** package.
- An iterator object can iterate the elements of collections ***only in a forward direction.***

Java Collection : Iterator

- **Getting an Iterator**

  The iterator() method can be used to get an Iterator for any collection (ArrayList, LinkedList etc.).

```java
// Import the ArrayList class and the Iterator class
import java.util.ArrayList;
import java.util.Iterator;

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

    // Make a collection
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    // Get the iterator
    Iterator<String> it = cars.iterator();

    // Print the first item
    System.out.println(it.next());
  }
}
```

Java Collection : Iterator

- **Looping Through a Collection**
  To loop through a collection, use the hasNext() and next() method of the Iterator:

Example

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

Java Collection : Iterator

- **Removing Items from a Collection**

  Iterators are designed to easily change the collections that they loop through. The remove() method can remove items from a collection while looping**.**

Note:

Trying to remove items using a for loop or a for-each loop would not work correctly.

Because the collection is changing size at the same time that the code is trying to loop.

Use an iterator to remove numbers less than 10 from a collection:

```
import java.util.ArrayList;
import java.util.Iterator;
public class MyClass {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(12);
    numbers.add(8);
    numbers.add(2);
    numbers.add(23);
    Iterator<Integer> it = numbers.iterator();
    while(it.hasNext()) {
      Integer i = it.next();
      if(i < 10) {
        it.remove();
      }
    }
    System.out.println(numbers);
  }
}
```

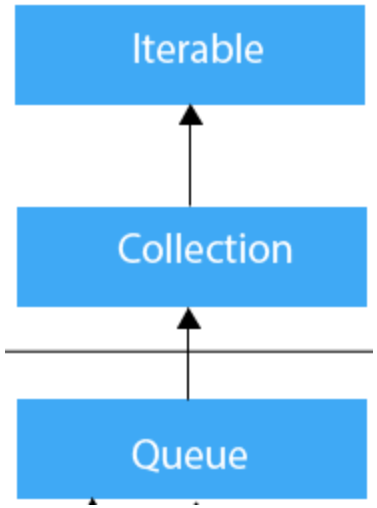
Collection Interface

Class java.util.Collections

- This class consists exclusively of static methods that operate on or return collections

Collection : Collection Interface

- **Collection Interface**
  - The Collection interface is the interface which is implemented by all the classes in the collection framework.
  - It declares the methods that every collection will have.
  - In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.



Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

Collection Interface : Methods

- **Collection Interface : Popular methods**

Name	Descriptions
binarySearch(List list, Object key, Comparator c)	Searches the specified List for the specified Object using the binary search algorithm.
copy(List dest, List src)	Copies all of the elements from one List into another.
fill(List lit, Object o)	Replaces all of the elements of the specified List with the specified element.
max(Collection coll)	Returns the maximum element of the given Collection, according to the natural ordering of its elements.
min(Collection coll)	Returns the minimum element of the given Collection, according to the natural ordering of its elements.
shuffle(List list)	Randomly permutes the specified list using a default source of randomness.
reverse(List l)	Reverses the order of the elements in the specified List.
sort(List list)	Sorts the specified List into ascending order, according to the natural ordering of its elements

Collection Interface: Example

```
import java.util.*;
public class Coll {
    public static void main(String[] args) {
        List <Integer> al=new ArrayList<Integer>();
            al.add(10);
            al.add(20);
            al.add(30);
          System.out.println(al);
          Collections.addAll(al,100,5,0,1,50);
          System.out.println(al);
          Collections.sort(al);
          //Collections.sort(al,Collections.reverseOrder());
          System.out.println(al);
          Collections.reverse(al);
          System.out.println(al);
          System.out.println(Collections.max(al));
          System.out.println(Collections.min(al));
          Collections.shuffle(al);
          System.out.println(al);
```

```
[10, 20, 30]
[10, 20, 30, 100, 5, 0, 1, 50]
[0, 1, 5, 10, 20, 30, 50, 100]
[100, 50, 30, 20, 10, 5, 1, 0]
100
0
[50, 5, 30, 10, 20, 1, 100, 0]
```

List Interface

ArrayList, LinkedList, Vector & Stack

Ordered collection , Index based methods

Allow Duplicate and Null elements

How to create List

- List in Java provides the facility to maintain the ordered collection.
- It contains the index-based methods to insert, update, delete and search the elements.
- It can have the duplicate elements also.
- We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

```
//Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();

//Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();

//Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();

//Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

List is an interface whereas ArrayList is the implementation class of List.

Java Collection

ArrayList

resizable array

- elements can be added and removed from an ArrayList whenever you want

Java Collection : ArrayList

- **Java ArrayList**

The ArrayList class is a resizable array, which can be found in the java.util package.

- **Array Vs. ArrayList**

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different:

Create an `ArrayList` object called **cars** that will store strings:

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Java Collection : ArrayList

- **Java ArrayList : *Add Items***

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

```
import java.util.ArrayList;

public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

ArrayList - Operations

- **Java ArrayList : *Access an Item***

To access an element in the ArrayList, use the get() method and refer to the index number: ***cars.get(0);***

Remember: *Array indexes start with 0: [0] is the first element. [1] is the second element, etc.*

- **Java ArrayList : *Change an Item***

To modify an element, use the set() method and refer to the index number:

cars.set(0, "Opel");

- **Java ArrayList :** *Remove an Item*

To remove an element, use the remove() method and refer to the index number:

cars.remove(0);

To ***remove all*** *the elements in the ArrayList, use the clear() method :* ***cars.clear()***

- **Java ArrayList : *ArrayList Size***

To find out how many elements an ArrayList have, use the size method:

cars.size();

ArrayList - Loop Through

- **Java ArrayList : *loop through***

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

```
public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

*You can also loop through an ArrayList with the **for-each** loop:*

```
for (String i : cars) {
  System.out.println(i);
}
```

ArrayList – Various type

- **Java ArrayList : Other type**

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type).

To use other types, such as int, you must specify an equivalent wrapper class: Integer.

Create an `ArrayList` to store numbers (add elements of type `Integer`):

```
import java.util.ArrayList;

public class MyClass {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
```

For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc.:

ArrayList - Sort an ArrayList

- **Java ArrayList : *Sort an ArrayList of String***

Another useful class in the java.util package is the ***Collections class***, which include the sort() method for sorting lists alphabetically or numerically:

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    Collections.sort(cars);  // Sort cars
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

ArrayList - Sort an ArrayList

- **Java ArrayList :** ***Sort an ArrayList of Integer***

```
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class MyClass {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

LinkedList

containers

- The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list.

Java Collection : LinkedList

- **Java LinkedListt :**

In the previous chapter, you learned about the ArrayList class. The LinkedList class is almost identical to the ArrayList:

```
// Import the LinkedList class
import java.util.LinkedList;

public class MyClass {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Java Collection : LinkedList

ArrayList vs. LinkedList

- The LinkedList class is a collection which can contain many objects of the same type, just like the ArrayList.
- The LinkedList class has all of the same methods as the ArrayList class because they both implement the List interface. This means that you can add items, change items, remove items and clear the list in the same way.
- However, while the ArrayList class and the LinkedList class can be used in the same way, they are built very differently.

Java Collection : LinkedList

ArrayList and LinkedList: How they work?

- **How the ArrayList works**

  The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

- **How the LinkedList works**

  The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

Java Collection : LinkedList

ArrayList and LinkedList: When To Use them?

- **It is best to use an ArrayList when:**
  - You want to access random items frequently
  - You only need to add or remove elements at the end of the list
- **It is best to use a LinkedList when:**
  - You only use the list by looping through it instead of accessing random items
  - You frequently need to add and remove items from the beginning or middle of the list

Java Collection : LinkedList

ArrayList and LinkedList: LinkedList Methods

- For many cases, the ArrayList is more efficient as it is common to need access to random items in the list, but the LinkedList provides several methods to do certain operations more efficiently:

Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

LinkedList : Example

```
public class LL{
    public static void main(String[] args) {
        //List <String> ll=new LinkedList<String>();
        LinkedList <String> ll=new LinkedList<String>();
            ll.add("Sakib");
            ll.add("Akib");
            ll.add("Galib");
          System.out.println(ll);
          Collections.sort(ll);
          System.out.println(ll);
            ll.addFirst("Rakiba");
            ll.addLast("Meherin");
          Iterator it=ll.iterator();
          while(it.hasNext())
              System.out.println(it.next());
          for(String x:ll)
            System.out.print(x+ " ");
    }
}
```

```
package coll.list;

import java.util.Iterator;
import java.util.Collections;
import java.util.List;
import java.util.LinkedList;
```

Output:

```
[Sakib, Akib, Galib]
[Akib, Galib, Sakib]
Rakiba
Akib
Galib
Sakib
Meherin
Rakiba Akib Galib Sakib Meherin
```

Vector

Dynamic Array

- Vector is like the dynamic array which can grow or shrink its size, there is no size limit.

Java Collection : 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 recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

[

It is similar to the ArrayList, but with two differences-

- ✓ Vector is synchronized.
- ✓ Java Vector contains many legacy methods that are not the part of a collections framework.

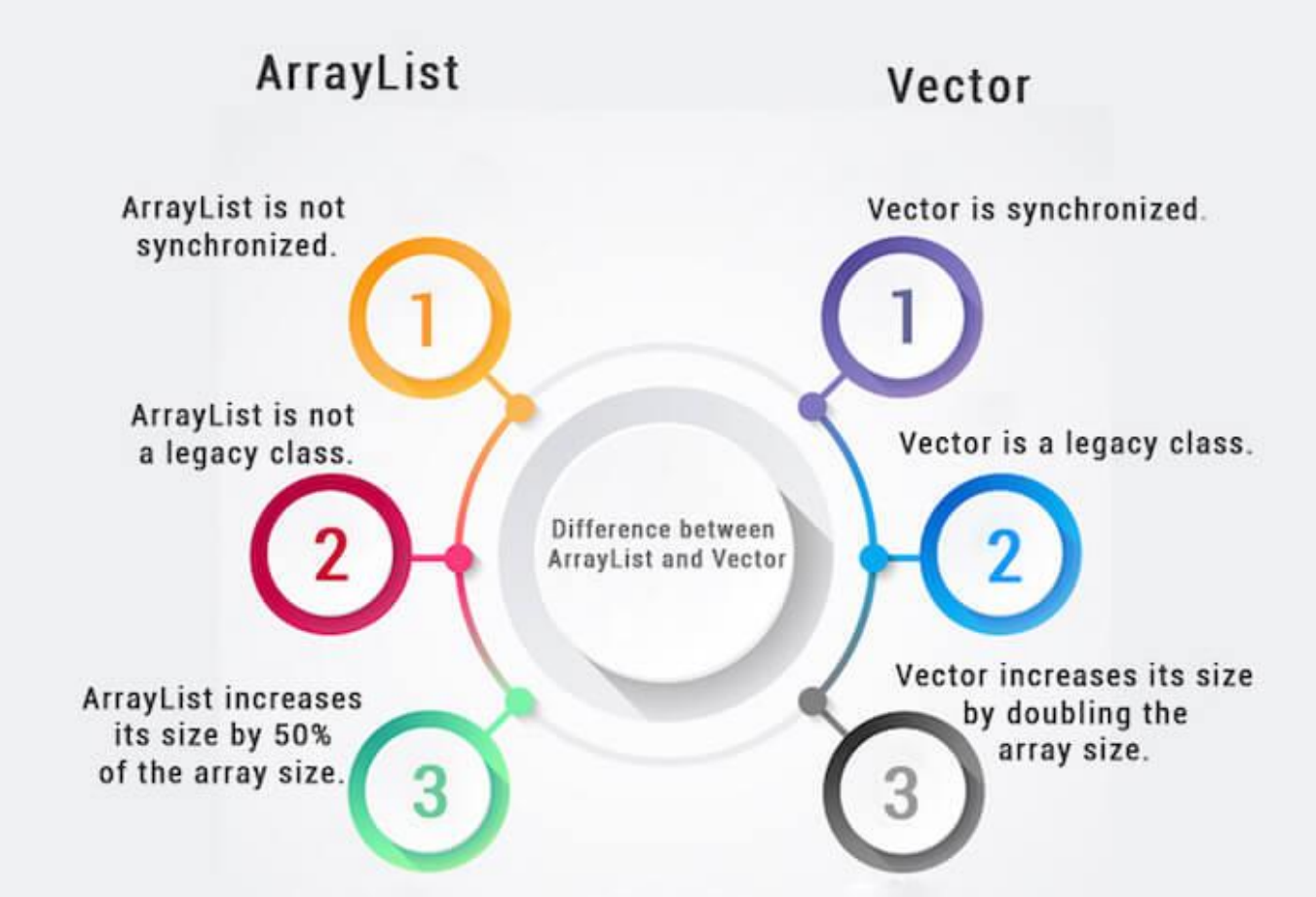
Java Collection : Vector

ArrayList and Vector both implements List interface and maintains insertion order. However, there are many differences between ArrayList and Vector classes that are given below.

ArrayList	Vector
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the **Iterator** interface to traverse the elements.	A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements.

Java Collection : Vector

ArrayList and Vector both implements List interface and maintains insertion order. However, there are many differences between ArrayList and Vector classes that are given below.



Collection : Vector - Example

```
package coll.list;
import java.util.Collections;
import java.util.Vector;
public class Vec {
    public static void main(String args[]){
        Vector<Integer> v=new Vector<Integer>();
                v.add(10);
                v.add(20);
                v.add(30);
                v.add(0,5);
                v.set(2,50);
        System.out.println(v);
        Collections.sort(v);
        System.out.println(v);
         System.out.println("Size is: "+v.size());
         System.out.println("Default capacity is: "+v.capacity());
    }
}
```

Output:

```
[5, 10, 50, 30]
[5, 10, 30, 50]
Size is: 4
Default capacity is: 10
```

Stack

Last-In-First-Out (LIFO)

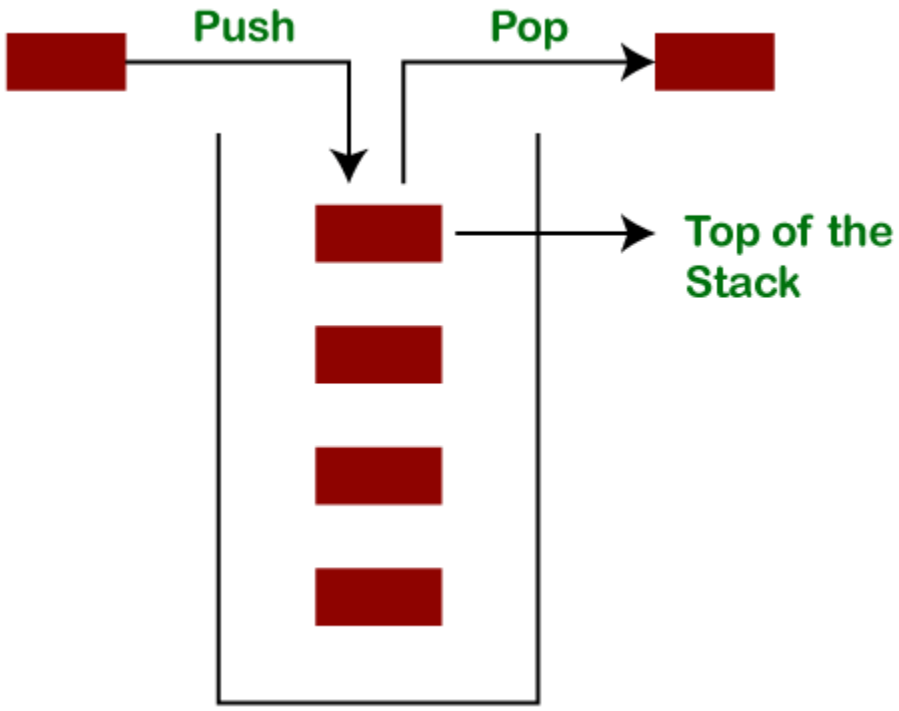
- The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO).

Java Collection : Stack

- The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO).

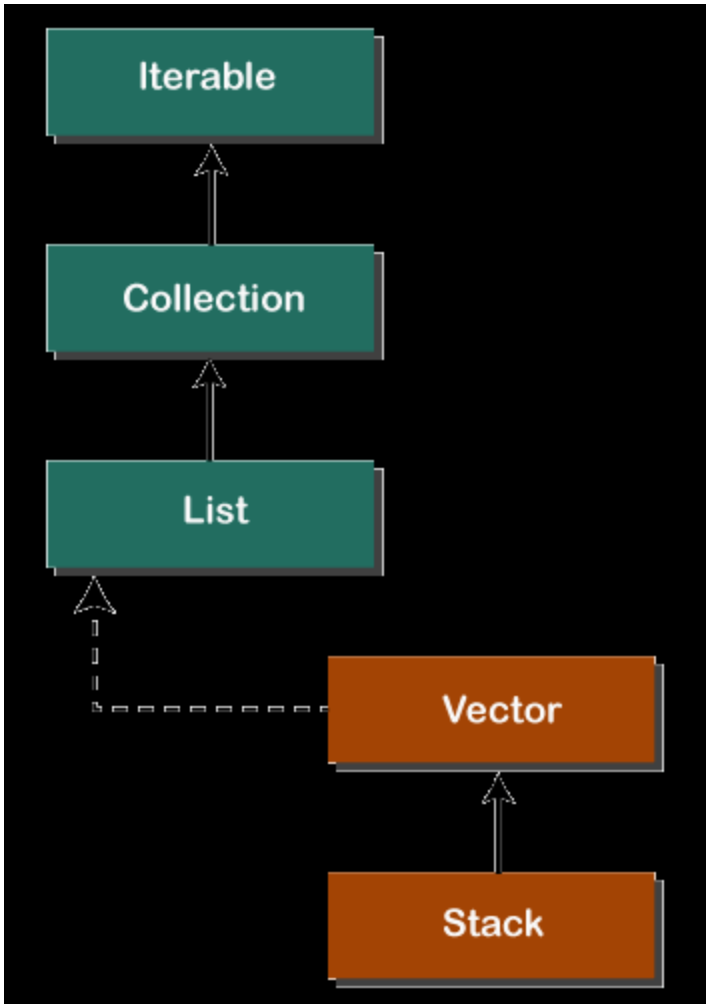
- Stack class in java provides different operations such as push, pop, search, etc.

- The stack data structure has the two most important operations that are push and pop. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.



Java Collection : Stack

- In Java, Stack is a class that falls under the Collection framework that extends the Vector class. It also implements interfaces List, Collection, Iterable, Cloneable, Serializable.
- It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package.
- The stack class arranged in the Collections framework hierarchy, as shown below.



Collection : Stack - Example

```
package coll.list;
import java.util.*;
import java.util.Collections;
import java.util.Stack;
public class St {
    public static void main(String args[]){
    Stack<Integer> s=new Stack<Integer>();
                    s.push(10);
                    s.push(5);
                    s.push(100);
                    s.push(50);
        System.out.println(s);
        Collections.sort(s);
        System.out.println(s);
        int top = s.peek();
       System.out.println("Element at top: " + top);
       System.out.println(s.pop());
    }
}
```

```
[10, 5, 100, 50]
[5, 10, 50, 100]
Element at top: 100
100
```

Queue Interface

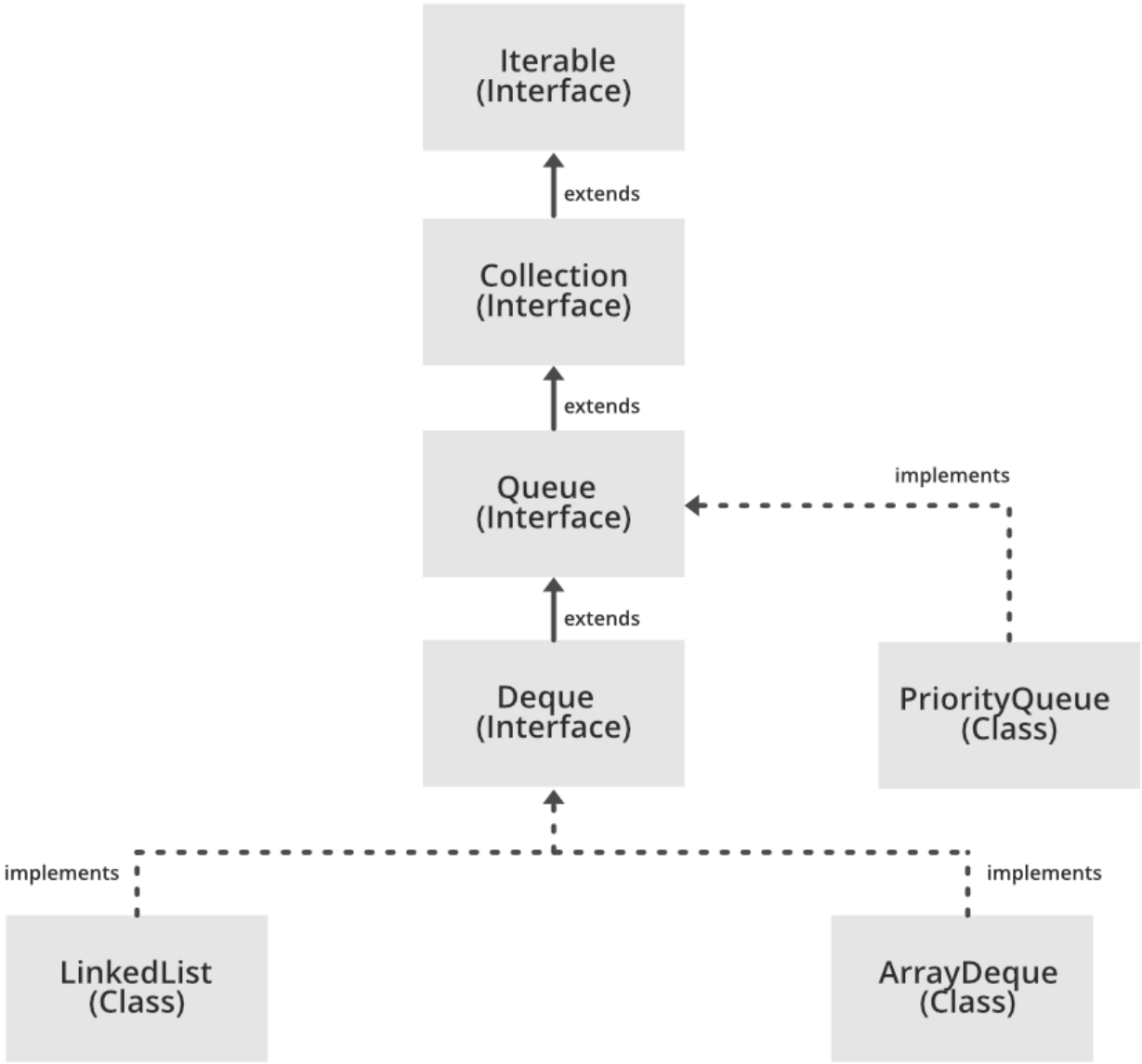
First In First Out (FIFO)

- Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

Collection: Queue Interface

- The Queue interface present in the java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order.
- It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.
- Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.It is to be noted that both the implementations are not thread safe.
  - PriorityBlockingQueue is one alternative implementation if thread safe implementation is needed.

Queue Interface Hierarchy



Methods of Queue Interface

METHOD	DESCRIPTION
add(element)	This method is used to add elements at the tail of queue. More specifically, at the last of linked-list if it is used, or according to the priority in case of priority queue implementation.
element()	This method is similar to peek(). It throws NoSuchElementException when the queue is empty.
offer(element)	This method is used to insert an element in the queue. This method is preferable to add() method since this method does not throws an exception when the capacity of the container is full since it returns false.
peek()	This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
poll()	This method removes and returns the head of the queue. It returns null if the queue is empty.
remove()	This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty.

PriorityQueue

First In First Out (FIFO)

- Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

PriorityQueue : Example

- **PriorityQueue class**

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

- **PriorityQueue class declaration**

Let's see the declaration for java.util.PriorityQueue class.

public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

PriorityQueue : Example

```
// Java program to add elements
// to a Queue

import java.util.*;

public class GFG {

    public static void main(String args[])
    {
        Queue<String> pq = new PriorityQueue<>();

        pq.add("Geeks");
        pq.add("For");
        pq.add("Geeks");

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

PriorityQueue : Example

```java
// Java program to remove elements
// from a Queue

import java.util.*;

public class GFG {

    public static void main(String args[])
    {
        Queue<String> pq = new PriorityQueue<>();

        pq.add("Geeks");
        pq.add("For");
        pq.add("Geeks");

        System.out.println("Initial Queue " + pq);

        pq.remove("Geeks");

        System.out.println("After Remove " + pq);

        System.out.println("Poll Method " + pq.poll());

        System.out.println("Final Queue " + pq);
    }
}
```

Output:

```
Initial Queue [For, Geeks, Geeks]
After Remove [For, Geeks]
Poll Method For
Final Queue [Geeks]
```

PriorityQueue : Example

```
// Java program to iterate elements
// to a Queue

import java.util.*;

public class GFG {

    public static void main(String args[])
    {
        Queue<String> pq = new PriorityQueue<>();

        pq.add("Geeks");
        pq.add("For");
        pq.add("Geeks");

        Iterator iterator = pq.iterator();

        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

PriorityQueue : Example

```java
// Java program to demonstrate the
// creation of queue object using the
// LinkedList class
import java.util.*;
class GfG {
    public static void main(String args[])
    {
        // Creating empty LinkedList
        Queue<Integer> ll
            = new LinkedList<Integer>();

        // Adding items to the ll
        // using add()
        ll.add(10);
        ll.add(20);
        ll.add(15);
        // Printing the top element of
        // the LinkedList
        System.out.println(ll.peek());
        // Printing the top element and removing it
        // from the LinkedList container
        System.out.println(ll.poll());
        // Printing the top element again
        System.out.println(ll.peek());
    }
}
```

Output:

```
10
10
20
```

Deque Interface

Double ended queue

insertion and removal at both end

Collection: Deque Interface

- Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for "double ended queue".
- Deque Interface declaration

public interface Deque<E> extends Queue<E>

Methods of Java Deque Interface

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

ArrayDeque

deque and resizable-array

insertion and removal at both end

ArrayDeque is faster than LinkedList and Stack.

ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

ArrayDeque : Example

```
import java.util.*;
public class ArrayDequeExample {
   public static void main(String[] args) {
   //Creating Deque and adding elements
   Deque<String> deque = new ArrayDeque<String>();
   deque.add("Ravi");
   deque.add("Vijay");
   deque.add("Ajay");
   //Traversing elements
   for (String str : deque) {
   System.out.println(str);
   }
   }
}
```

Output:

```
Ravi
Vijay
Ajay
```

ArrayDeque : Example

- **Java ArrayDeque Example: offerFirst() and pollLast()**

```
Deque<String> deque=new ArrayDeque<String>();
deque.offer("arvind");
deque.offer("vimal");
deque.add("mukul");
deque.offerFirst("jai");
System.out.println("After offerFirst Traversal...");
for(String s:deque){
   System.out.println(s);
}
//deque.poll();
//deque.pollFirst();//it is same as poll()
deque.pollLast();
System.out.println("After pollLast() Traversal...");
for(String s:deque){
   System.out.println(s);
}
```

Output:

```
After offerFirst Traversal...
jai
arvind
vimal
mukul
After pollLast() Traversal...
jai
arvind
vimal
```

Map Interface

key : Value

- A HashMap however, store items in "key/value" pairs
  - ✓ String keys and Integer values or
  - ✓ Integer keys String values
  - ✓ String keys and String values

Collections : 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.

- **Uses:**

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

There are

- ***two interfaces*** for implementing Map in java: Map and SortedMap, and
- ***three classes***: HashMap, LinkedHashMap, and TreeMap.

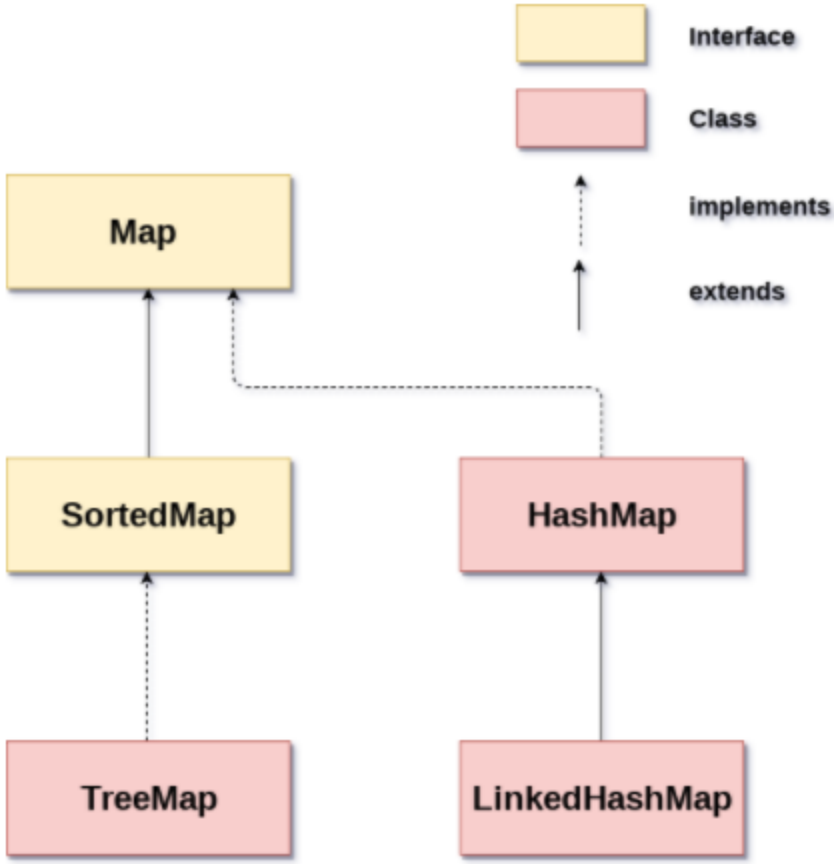


Fig: The hierarchy of Java Map

Collections : Map Interface

- 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.
- A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

Class	Description
HashMap	HashMap is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

HashMap

key, value pair, keys must unique

- A HashMap however, store items in "key/value" pairs
  - ✓ String keys and Integer values or
  - ✓ Integer keys String values
  - ✓ String keys and String values

Collection : HashMap

- **Points to remember**
  - Java HashMap contains values based on the key.
  - Java HashMap contains only unique keys.
  - Java HashMap may have one null key and multiple null values.
  - Java HashMap is non synchronized.
  - Java HashMap maintains no order.
  - The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

Java Collection : HashMap

Java HashMap: key : Value Pair

- In the ArrayList chapter, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (int type). A HashMap however, store items in "key/value" pairs, and you can access them by an index of another type (e.g. a String).

- One object is used as a key (index) to another object (value). It can store different types: String keys and Integer values, or the same type, like: String keys and String values:

Example

Create a `HashMap` object called **capitalCities** that will store `String` **keys** and `String` **values**

```
import java.util.HashMap; // import the HashMap class

HashMap<String, String> capitalCities = new HashMap<String, String>();
```

Java Collection : HashMap

Java HashMap: *Add Items*

- The HashMap class has many useful methods. For example, to add items to it, use the put() method:

```
// Import the HashMap class
import java.util.HashMap;

public class MyClass {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
  }
}
```

HashMap - Operations

- **Java HashMap :** ***Access an Item***

To access a value in the HashMap, use the get() method and refer to its key:

capitalCities.get("England");

- **Java HashMap :** ***Remove an Item***

To remove an item, use the remove() method and refer to the key:

capitalCities.remove("England");

To remove all items, use the clear() method: ***capitalCities.clear();***

- **Java HashMap :** ***HashSet Size***

To find out how many items there are, use the size method:

capitalCities.size();

HashMap : Example

```
package coll.map;
import java.util.HashMap;
public class HM {
    public static void main(String args[]){
        HashMap<String, String> hm=new HashMap<String, String>();
            hm.put("Bangladesh", "Dhaka");
            hm.put("India", "New Delhi");
            hm.put("Pakistan", "Islamabad");
        System.out.println(hm);
        System.out.println(hm.get("Bangladesh"));
        System.out.println(hm.remove("India"));
        System.out.println(hm);
        System.out.println(hm.size());
        System.out.println(hm);
        hm.clear();
        System.out.println(hm);
        System.out.println(hm.size());
    }
}
```

```
{Bangladesh=Dhaka, Pakistan=Islamabad, India=New Delhi}
Dhaka
New Delhi
{Bangladesh=Dhaka, Pakistan=Islamabad}
2
{Bangladesh=Dhaka, Pakistan=Islamabad}
{}
0
```

Java Collection : HashMap

Java HashMap: *Loop Through a HashMap*

- Loop through the items of a HashMap with a for-each loop.
- Note: Use the keySet() method if you only want the keys, and use the values() method if you only want the values:

```
// Print keys
for (String i : capitalCities.keySet()) {
  System.out.println(i);
}
```

```
// Print values
for (String i : capitalCities.values()) {
  System.out.println(i);
}
```

```
// Print keys and values
for (String i : capitalCities.keySet()) {
  System.out.println("key: " + i + " value: " + capitalCities.get(i));
}
```

HashMap : Example

```
package coll.map;
import java.util.HashMap;
public class HML {
    public static void main(String args[]){

        HashMap<Integer, String> hm=new HashMap<Integer, String>();
        hm.put(2, "Sakib");
        hm.put(3, "Miraz");
        hm.put(104, "Ahmed Nafiz");
        hm.put(22, "Rejoana");
        System.out.println(hm);

        for(int x : hm.keySet())
                System.out.println(x);
                //System.out.println(hm.get(x));
        for(String x : hm.values())
                System.out.println(x);
        for(int x : hm.keySet())
                System.out.println(x + " = "+ hm.get(x));
```

```
{2=Sakib, 3=Miraz, 22=Rejoana, 104=Ahmed Nafiz}
2
3
22
104
Sakib
Miraz
Rejoana
Ahmed Nafiz
2 = Sakib
3 = Miraz
22 = Rejoana
104 = Ahmed Nafiz
```

Java Collection : HashMap

Java HashMap: *Other type*

- Keys and values in a HashMap are actually objects. In the examples above, we used objects of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an ***equivalent wrapper class: Integer***.
- For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

Java Collection : HashMap

- Create a HashMap object called people that will store String keys and Integer values:

```
// Import the HashMap class
import java.util.HashMap;

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

    // Create a HashMap object called people
    HashMap<String, Integer> people = new HashMap<String, Integer>()


    // Add keys and values (Name, Age)
    people.put("John", 32);
    people.put("Steve", 30);
    people.put("Angie", 33);

    for (String i : people.keySet()) {
      System.out.println("key: " + i + " value: " + people.get(i));
    }
  }
}
```

Set Interface

Unique Value

every item is unique, and it is found in the java.util package

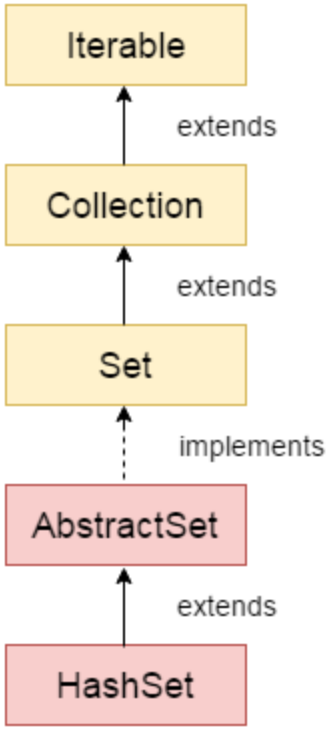
Collections : Set Interface

- **Difference between List and Set**

A list can contain duplicate elements whereas Set contains unique elements only.

- **Hierarchy of HashSet class**

- The HashSet class extends AbstractSet class which implements Set interface.
- The Set interface inherits Collection and Iterable interfaces in hierarchical order.



HashSet

Unique Value

- A HashSet is a collection of items where every item is unique, and it is found in the java.util package

Java Collection : HashSet

Java HashSet:

- A HashSet is a collection of items where every item is unique, and it is found in the java.util package:

Example

Create a `HashSet` object called **cars** that will store strings:

```
import java.util.HashSet; // Import the HashSet class

HashSet<String> cars = new HashSet<String>();
```

Java Collection : HashSet

Java HashSet : *Add Items*

- The HashSet class has many useful methods. For example, to add items to it, use the add() method:

```
// Import the HashSet class
import java.util.HashSet;

public class MyClass {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Note: In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

HashSet : Example

```
package coll.set;
import java.util.HashSet;

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

        HashSet<String> hs=new HashSet<String>();
            hs.add("Sakib");
            hs.add("Galib");
            hs.add("Nusrat");
            hs.add("Tazrian");
            hs.add("Sakib");
        System.out.println(hs);

         for (String x : hs)
           System.out.println(x);
    }
}
```

Output:

```
[Sakib, Nusrat, Galib, Tazrian]
Sakib
Nusrat
Galib
Tazrian
```

HashSet : Example

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet();
          set.add("One");
          set.add("Two");
          set.add("Three");
          set.add("Four");
          set.add("Five");
          Iterator<String> i=set.iterator();
          while(i.hasNext())
          {
          System.out.println(i.next());
          }
```

```
import java.util.*;
class HashSet2{
 public static void main(String args[]){
  //Creating HashSet and adding elements
  HashSet<String> set=new HashSet<String>();
  set.add("Ravi");
  set.add("Vijay");
  set.add("Ravi");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
Ajay
Vijay
Ravi
```

HashSet - Operations

- **Java HashSet:** ***Check If an Item Exists***

To check whether an item exists in a HashSet, use the contains() method:

cars.contains("Mazda");

- **Java HashSet:** ***Remove an Item***

To remove an item, use the remove() method:

cars.remove("Volvo");

To remove all items, use the clear() method: ***cars.clear();***

- **Java HashSet:** ***HashSet Size***

To find out how many items there are, use the size method:

cars.size();

Java Collection : HashSet

Java HashSet: ***Loop Through a HashSet***

- Loop through the items of an HashSet with a for-each loop:

```
for (String i : cars) {
  System.out.println(i);
}
```

Java Collection : HashSet

Java HashSet : ***Other type***

- Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type).
- To use other types, such as int, you must specify ***an equivalent wrapper class: Integer.***
- For other primitive types, use: Boolean for boolean, Character for char, Double for double, etc:

Java Collection : HashSet

- Use a HashSet that stores Integer objects:

```
import java.util.HashSet;

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

    // Create a HashSet object called numbers
    HashSet<Integer> numbers = new HashSet<Integer>();

    // Add values to the set
    numbers.add(4);
    numbers.add(7);
    numbers.add(8);

    // Show which numbers between 1 and 10 are in the set
    for(int i = 1; i <= 10; i++) {
      if(numbers.contains(i)) {
        System.out.println(i + " was found in the set.");
      } else {
        System.out.println(i + " was not found in the set.");
      }
    }
  }
}
```