FPT UNIVERSITY

ĐẠI HỌC FPT CẦN THƠ

Collections

Session 9

Objectives

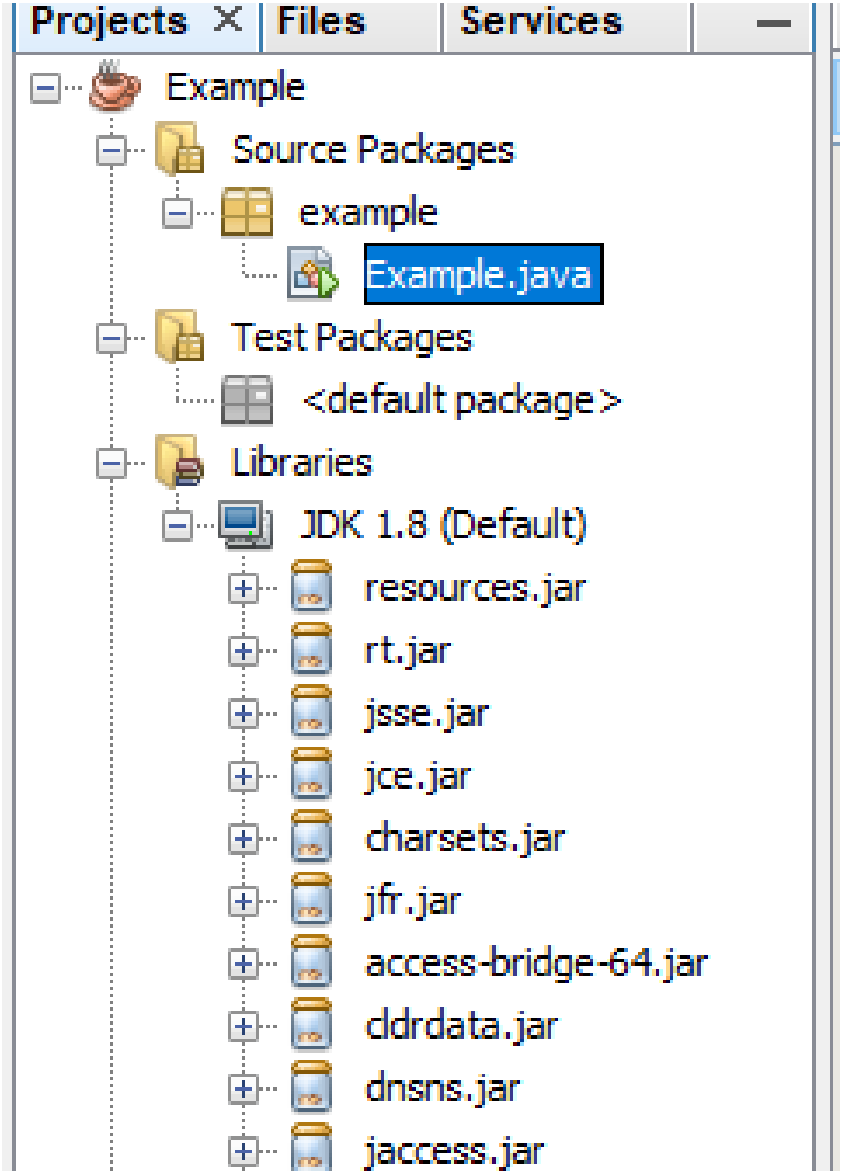
1. Introduction.
2. Interfaces.
3. Pre-defined Classes

Collections

- called a container — is simply an object that groups multiple elements into a single unit.
- used to store, retrieve, manipulate, and communicate aggregate data

What Is a Collections Framework?

- Framework is often a layered structure indicating what kind of programs can or should be built and how they would interrelate.



What Is a Collections Framework?

- The Collections Framework represents and manipulates collections.
- It includes the following:
  - Interfaces
  - Implementations
  - Algorithms

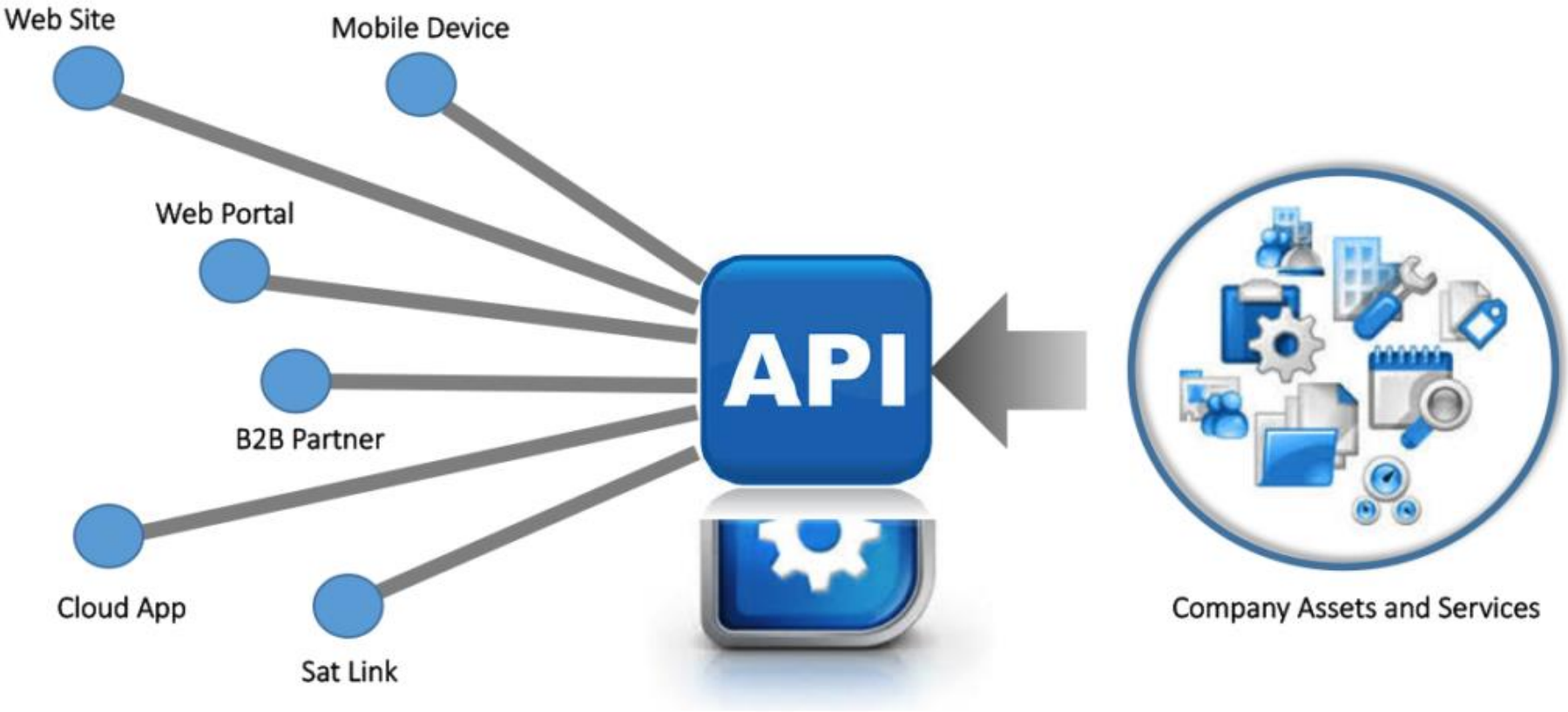
Benefits

- **Reduces programming effort** by providing useful data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.

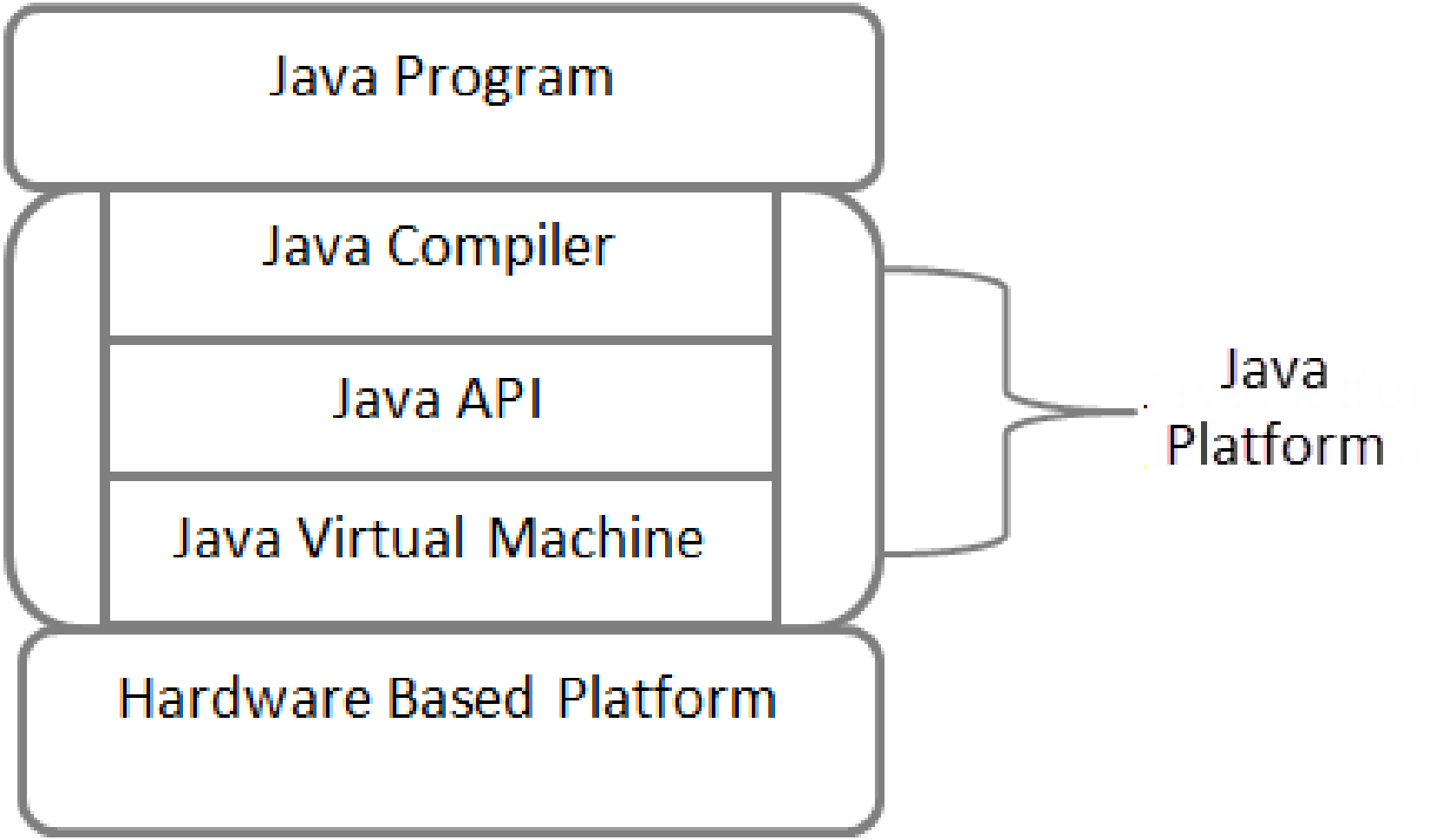
Benefits

- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

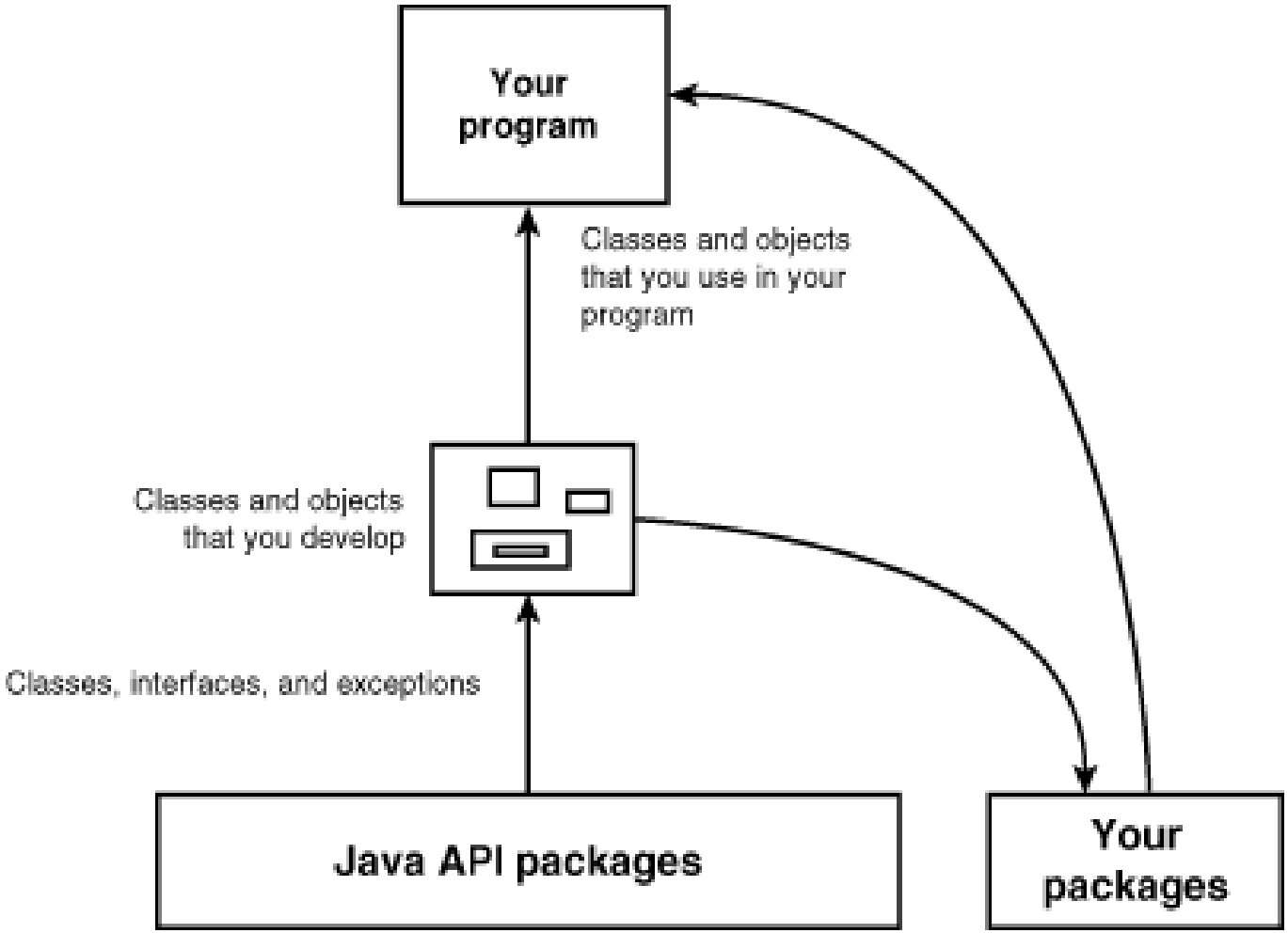
Application Programming Interface



API in Java


Java Program
Java Compiler
Java API
Java Virtual Machine
Hardware Based Platform
Java
Platform

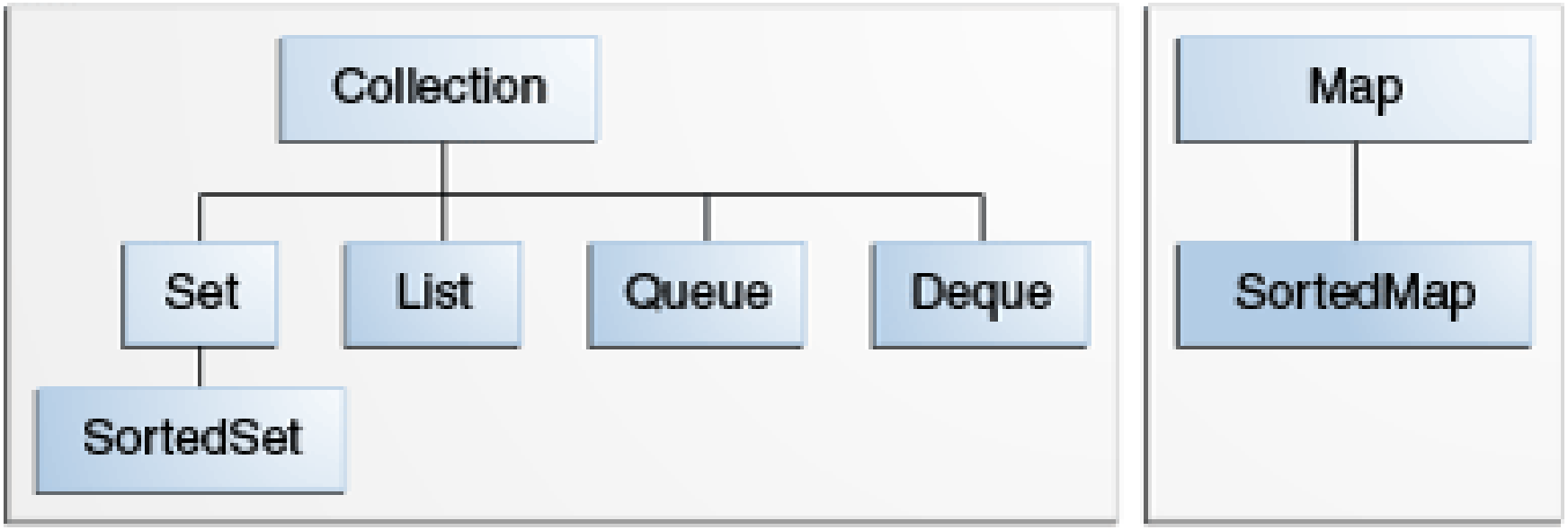
API in Java


Your program
Classes and objects that you use in your program
Classes and objects that you develop
Classes, interfaces, and exceptions
Java API packages
Your packages

Collection Interface

- Collections Framework consists of interfaces and classes for working with group of objects.
- At the top of the hierarchy, Collection interface lies.
- The Collection interface helps to convert the collection's type.
- The Collection interface is extended by the following sub interfaces:
  - Set
  - List
  - Queue
- Some of the Collection classes are as follows:
  - HashSet
  - LinkedHashSet
  - TreeSet

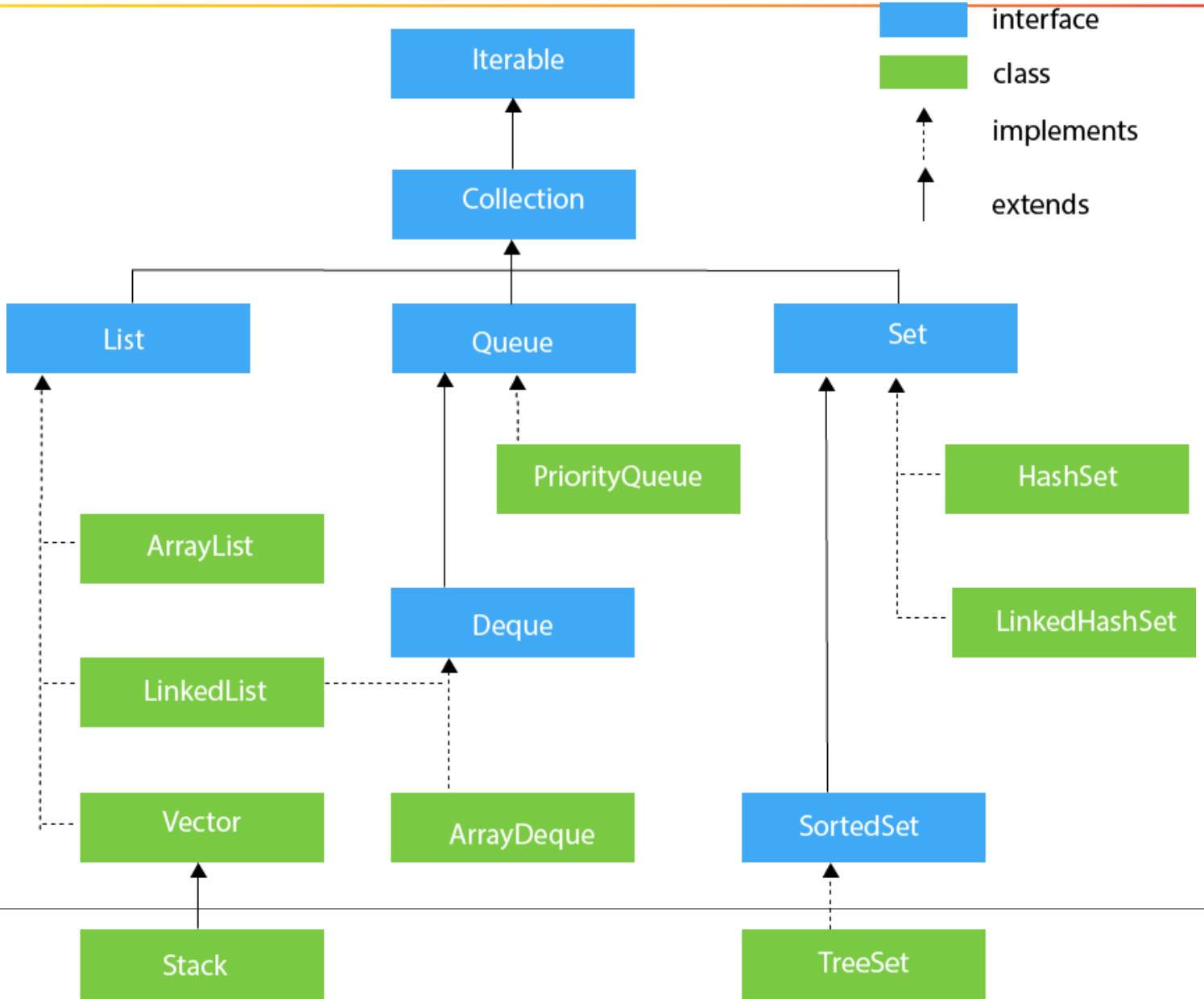
Collection Interface



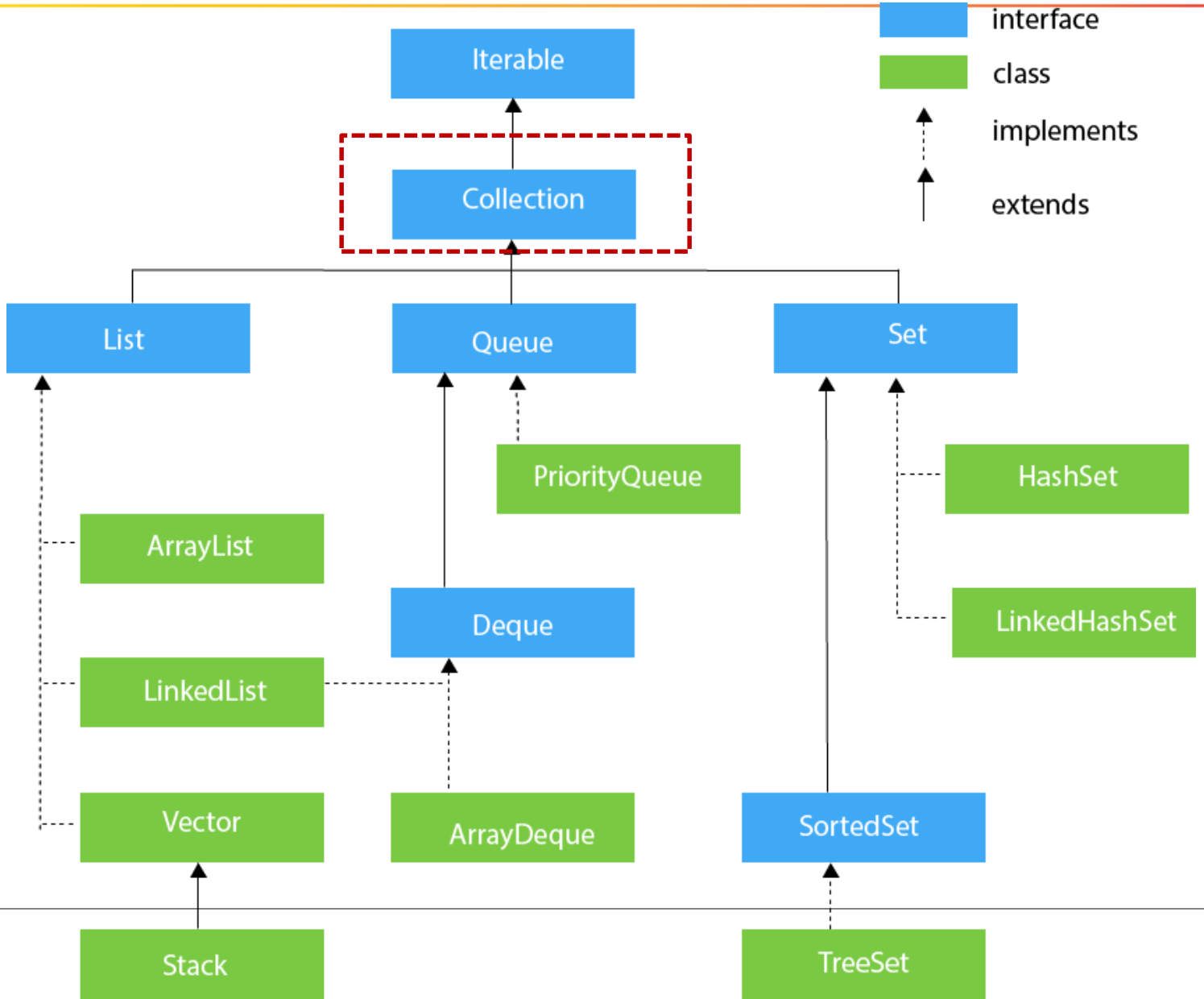
java.util Package

- The java.util package contains the definition of a number of useful classes providing a broad range of functionality.
- The package mainly contains collection classes that are useful for working with groups of objects.
- The package also contains the definition of classes that provides date and time facilities and many other utilities, such as calendar and dictionary.
- It also contains a list of classes and interfaces to manage a collection of data in memory.

The java.util package



The java.util package



Methods of Collection Interface

Method	Description
`add(Object x)`	Adds x to this collection
`addAll(Collection c)`	Adds every element of c to this collection
`clear()`	Removes every element from this collection
`contains(Object x)`	Returns true if this collection contains x
`containsAll(Collection c)`	Returns true if this collection contains every element of c
`isEmpty()`	Returns true if this collection contains no elements
`iterator()`	Returns an Iterator over this collection (see below)
`remove(Object x)`	Removes x from this collection
`removeAll(Collection c)`	Removes every element in c from this collection
`retainAll(Collection c)`	Removes from this collection every element that is not in c
`size()`	Returns the number of elements in this collection
`toArray()`	Returns an array containing the elements in this collection

Traversing Collections Using for-each construct

- This helps to traverse a collection or array using a for loop.
- The following Code Snippet illustrates the use of the for-each construct to print out each element of a collection on a separate line:

```
for (Object obj : collection)
    System.out.println(obj);
```

```
class Example{
    public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();
        list.add("Ravi");//Adding object in arraylist
        list.add("Vijay");
        list.add("Ravi");
        list.add("Ajay");
        for (String str : list) {
        System.out.println(str);
        }
    }
}
```

```
ıt - Example (run)
run:
Ravi
Vijay
Ravi
Ajay
BUILD SUCCESSFUL (total time: 0 seconds)
```

Traversing Collections Using Iterator

- These help to traverse through a collection.
- They also help to remove elements from the collection selectively.
- The iterator() method is invoked to obtain an Iterator for a collection.
- The Iterator interface includes the following methods:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
 void remove(); //optional
}
```

Traversing Collections Using Iterator

```
1   package example;
2   import java.util.*;
3   class Example{
4       public static void main(String args[]){
5           ArrayList<String> list=new ArrayList<String>();//Creating arraylist
6           list.add("Ravi");//Adding object in arraylist
7           list.add("Vijay");
8           list.add("Ravi");
9           list.add("Ajay");
10          //Traversing list through Iterator
11          Iterator itr=list.iterator();
12          while(itr.hasNext()){
13              System.out.println(itr.next());
14              }
15      }
16  }
```

Output - Example (run)

```
run:
Ravi
Vijay
Ravi
Ajay
BUILD SUCCESSFUL (total time: 0 seconds)
```

Bulk Operations

- Bulk operations perform shorthand operations on an entire Collection using the basic operations.
- The following table describes the methods for bulk operations:

Method	Description
`containsAll`	This method will return true if the target Collection contains all elements that exist in the specified Collection.
`addAll`	This method will add all the elements of the specified Collection to the target Collection.
`removeAll`	This method will remove all the elements from the target Collection that exist in the specified Collection.
`retainAll`	This method will remove those elements from the target Collection that do not exist in the specified Collection.

Bulk Operations

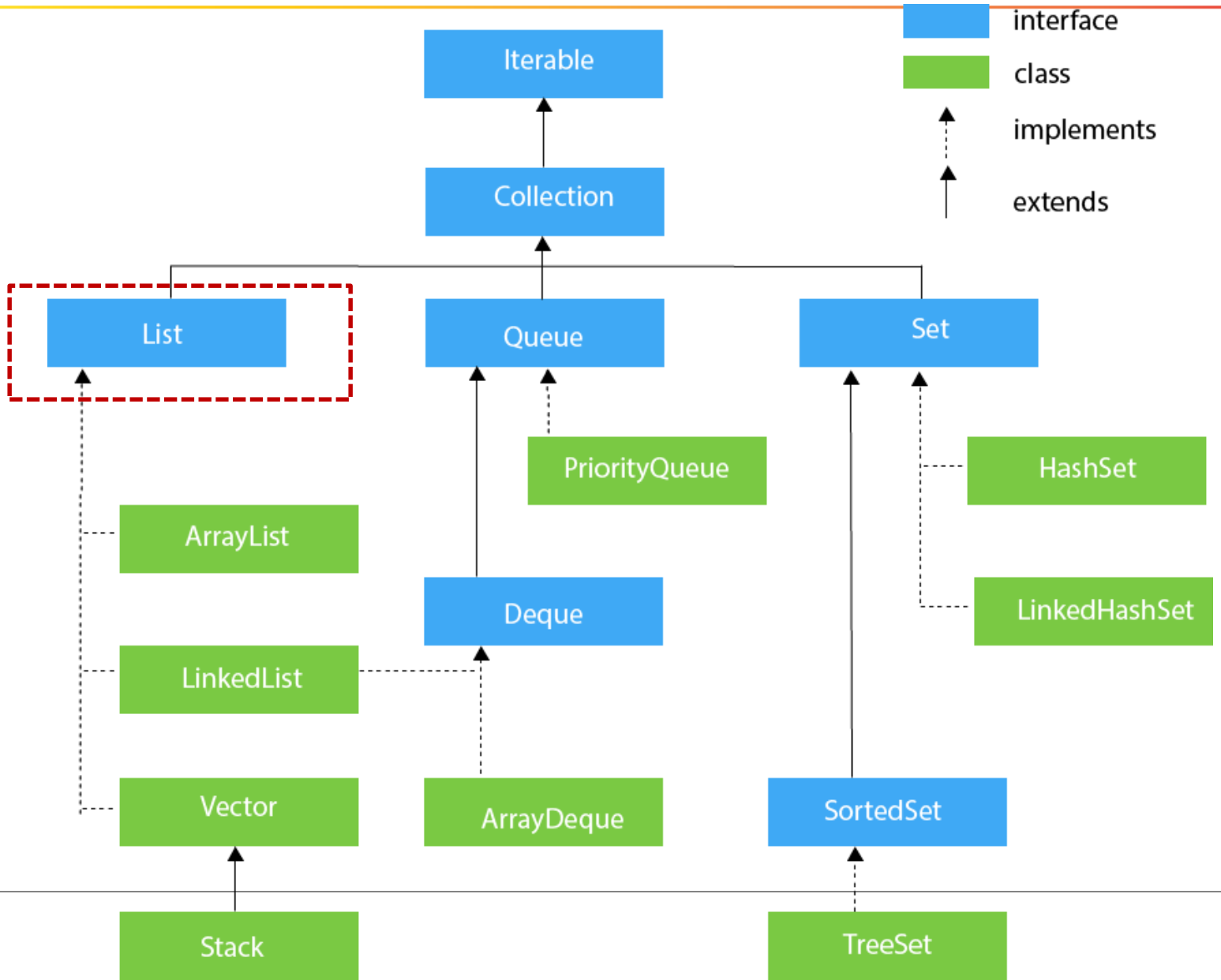
```
class Example{
    public static void main(String args[]){
        ArrayList<String> list1 = new ArrayList<String>();
        list1.add("Ravi");
        list1.add("Vijay");
        list1.add("Ravi");
        list1.add("Ajay");

        ArrayList<String> list2 = new ArrayList<String>();
        list2.addAll(list1);
        for (String str : list2) {
        System.out.println(str);
        }
    }
}
```

example.Example > main > for (String str : list2)

out - Example (run)

```
run:
Ravi
Vijay
Ravi
Ajay
BUILD SUCCESSFUL (total time: 0 seconds)
```

Lists


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Lists

- The `List` interface is an extension of the `Collection` interface.
- It defines an ordered collection of data and allows duplicate objects to be added to a list.
- Its advantage is that it adds position-oriented operations, enabling programmers to work with a part of the list.

Lists

- The `List` interface uses an index for ordering the elements while storing them in a list.
- List has methods that allow access to elements based on their position, search for a specific element, and return their position, in addition to performing arbitrary range operations.
- It also provides the `List` iterator to take advantage of its sequential nature.

Methods of List Interface

- `add(int index, E element)`
- `addAll(int index, Collection<? extends E> c)`
- `get(int index)`
- `set(int index, E element)`
- `remove(int index)`
- `subList(int start, int end)`
- `indexOf(Object o)`
- `lastIndexOf(Object o)`

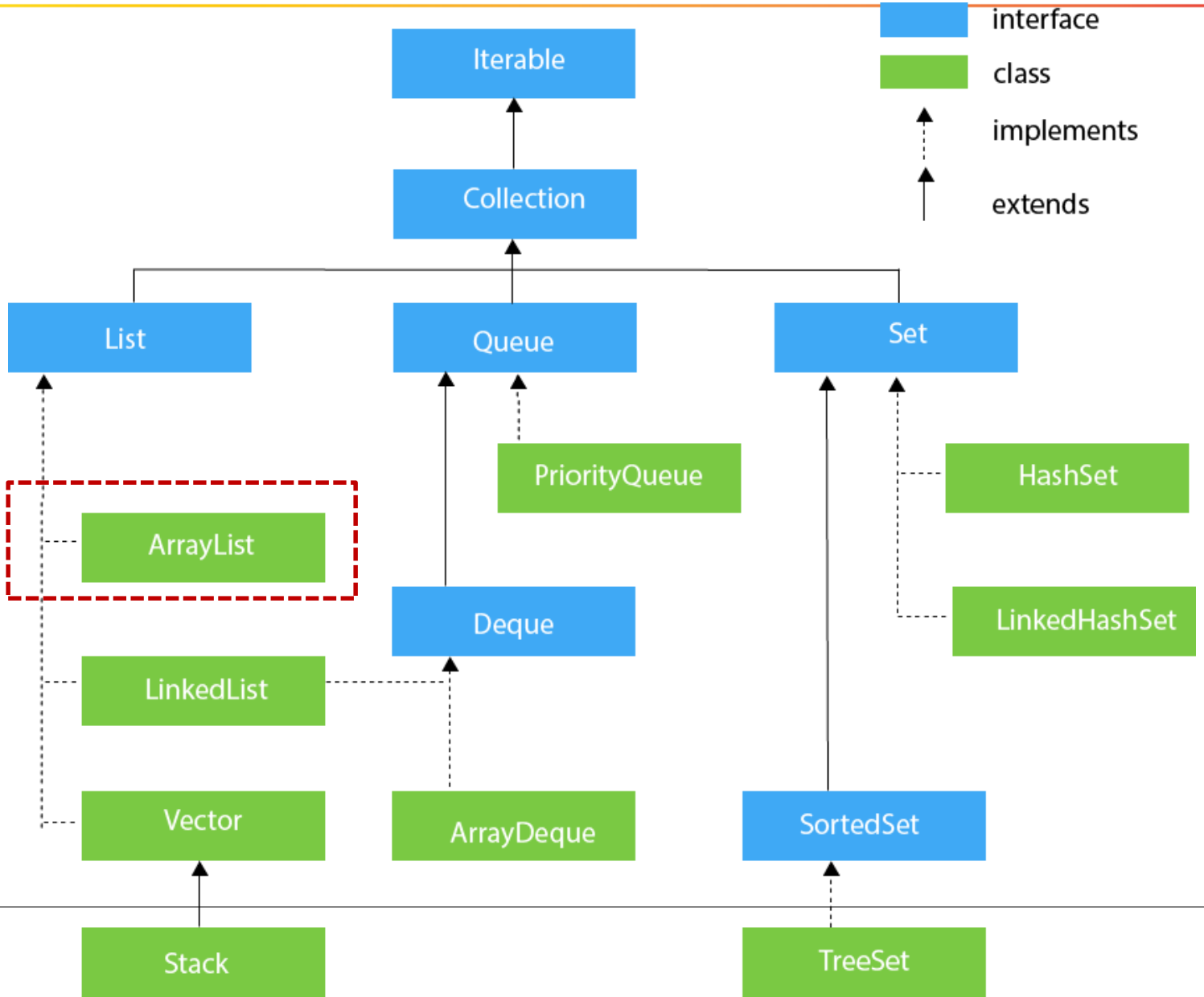
```
class Example{
    public static void main(String args[]){
        ArrayList<String> list1 = new ArrayList<String>();
        list1.add("Ravi");
        list1.add("Vijay");
        list1.add("Ravi");
        list1.add("Ajay");

        list1.remove(3);
        for (String str : list1) {
        System.out.println(str);
        }
    }
}
```

example.Example > main >

ut - Example (run) ×

```
run:
Ravi
Vijay
Ravi
BUILD SUCCESSFUL (total time: 0 seconds)
```

ArrayList Class



ArrayList Class

- `ArrayList` class is an implementation of the `List` interface in the `Collections` Framework.
- The `ArrayList` class creates a variable-length array of object references.
- The `ArrayList` class includes all elements, including null.
- In addition to implementing the methods of the `List` interface, this class provides methods to change the size of the array that is used internally to store the list.

ArrayList Class

- Each `ArrayList` instance includes a capacity that represents the size of the array.
- A capacity stores the elements in the list and grows automatically as elements are added to an `ArrayList`.
- `ArrayList` class is best suited for random access without inserting or removing elements from any place other than the end.

ArrayList Class

- An instance of ArrayList can be created using any one of the following constructors:
  - ArrayList()
  - ArrayList(Collection <? extends E> c)
  - ArrayList(int initialCapacity)

Methods of ArrayList Class

- `add(E obj)`
- `trimToSize()`
- `ensureCapacity(int minCap)`
- `clear()`
- `contains(Object obj)`
- `size()`

Methods of ArrayList Class

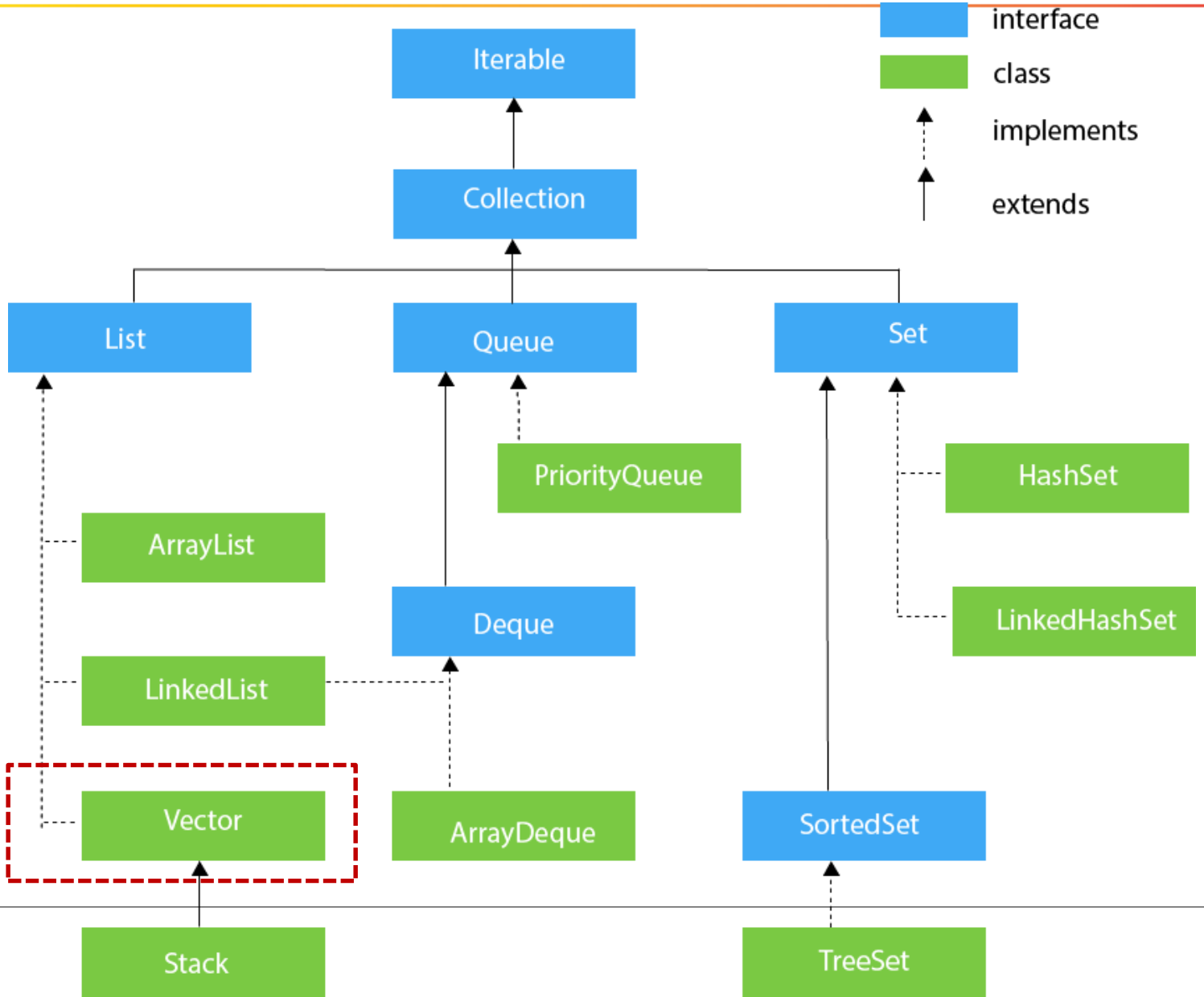
```
package example;
import java.util.*;
class Example{
    public static void main(String args[]){
        ArrayList<String> list1 = new ArrayList<String>();
        list1.add("Ravi");
        list1.add("Vijay");
        list1.add("Ravi");
        list1.add("Ajay");

        list1.clear();
        for (String str : list1) {
        System.out.println(str);
        }
    }
}
```

example.Example > main

ut - Example (run) ×

```
run:
BUILD SUCCESSFUL (total time: 0 seconds)
```

Vector Class



Vector Class

- The Vector class is similar to an ArrayList as it also implements dynamic array.
- Vector class stores an array of objects and the size of the array can increase or decrease.
- The elements in the Vector can be accessed using an integer index.
- Each vector maintains a capacity and a capacityIncrement to optimize storage management.
- The vector's storage increases in chunks specified by the capacityIncrement as components are added to it.
- The constructors of this class are as follows:
  - Vector()
  - Vector(Collection<? extends E> c)
  - Vector(int initCapacity)
  - Vector(int initCapacity, int capIncrement)

Methods of Vector Class

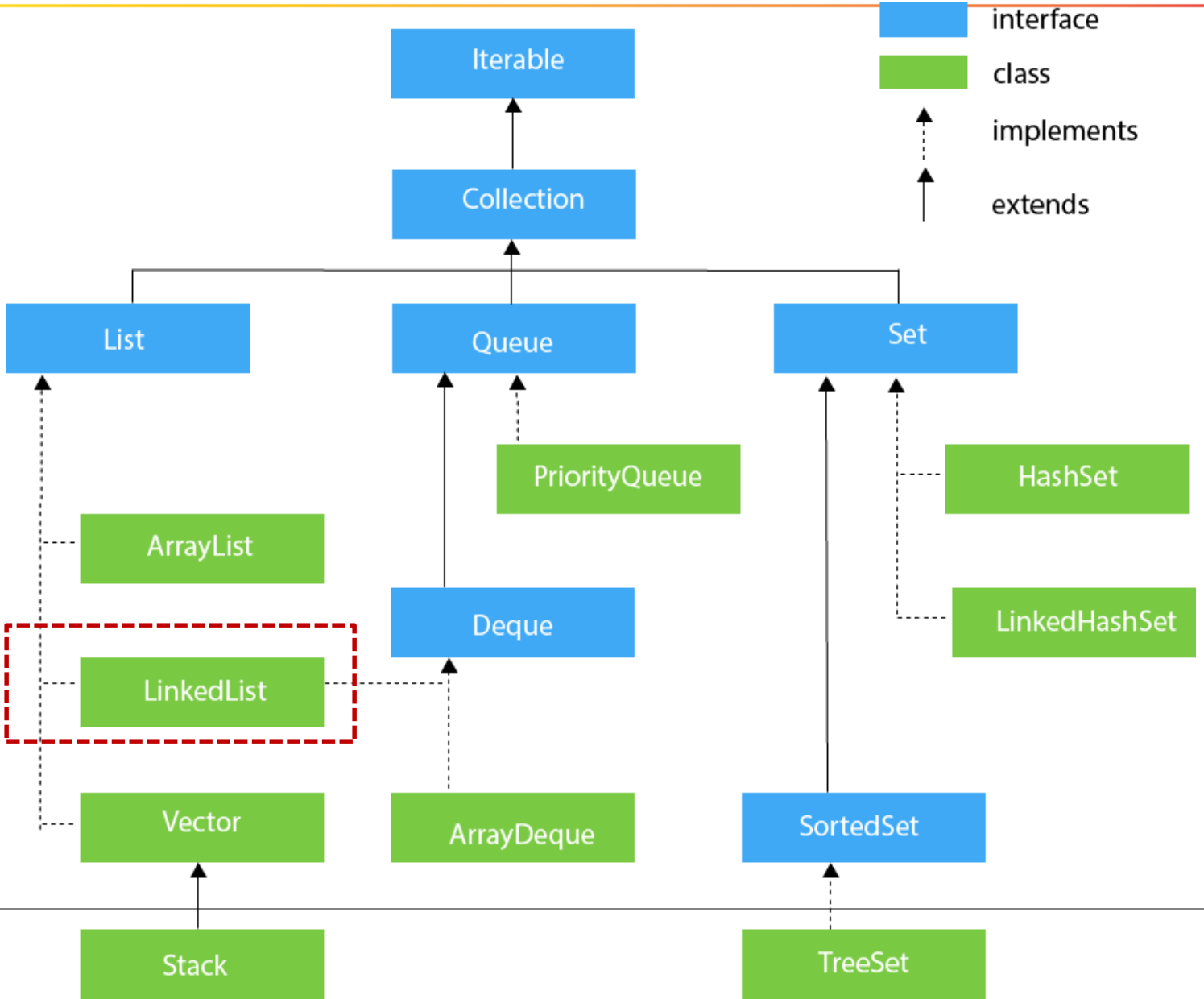
- `addElement(E obj)`
- `capacity()`
- `toArray()`
- `elementAt(int pos)`
- `removeElement(Object obj)`
- `clear()`

Methods of Vector Class

```
package example;
import java.util.*;
class Example{
    public static void main(String args[]){
        Vector<String> v=new Vector<String>();
        v.add("Ayush");
        v.add("Amit");
        v.add("Ashish");
        v.add("Garima");
        Iterator<String> itr=v.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

```
ut - Example (run)
run:
Ayush
Amit
Ashish
Garima
BUILD SUCCESSFUL (total time: 0 seconds)
```

LinkedList Class



LinkedList Class

- `LinkedList` class implements the `List` interface.
- An array stores objects in consecutive memory locations, whereas a linked list stores object as a separate link.
- A linked list is a list of objects having a link to the next object.
- There is usually a data element followed by an address element that contains the address of the next element in the list in a sequence.

LinkedList Class

- Each such item is referred as a node.
- Linked lists allow insertion and removal of nodes at any position in the list, but do not allow random access.
- There are several different types of linked lists - singly-linked lists, doubly-linked lists, and circularly-linked lists.
- Java provides the LinkedList class in the java.util package to implement linked lists.
  - LinkedList():
  - LinkedList(Collection <? extends E>c)

Methods of LinkedList Class

- `addFirst(E obj)`
- `addLast(E obj)`
- `getFirst()`
- `getLast()`
- `removeFirst()`
- `removeLast()`

Methods of LinkedList Class

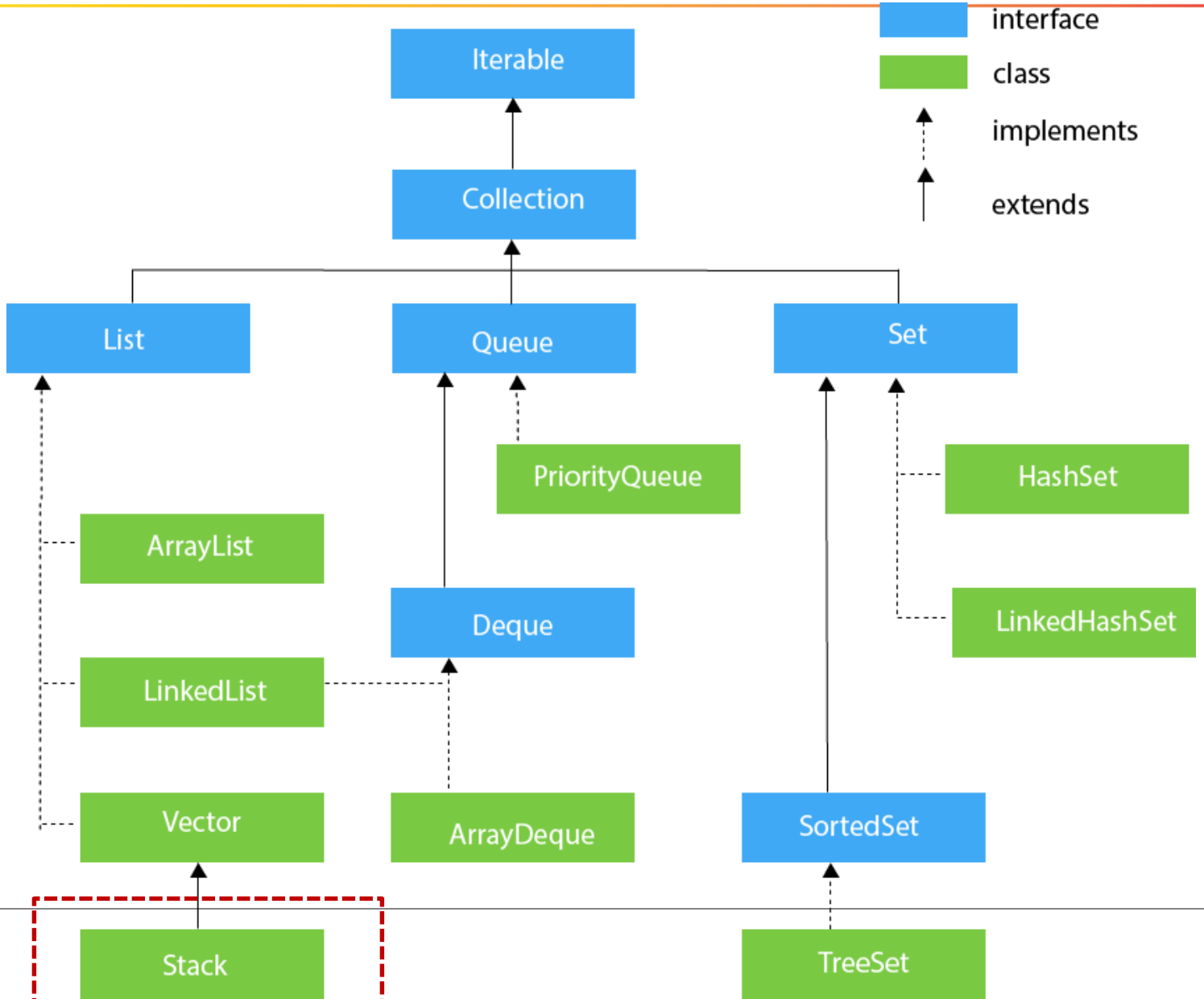
```
package example;
import java.util.*;
class Example{
    public static void main(String args[]){
        LinkedList<String> al=new LinkedList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");
        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
        System.out.println("last:"+al.getLast());
    }
}
```

example.Example > main >

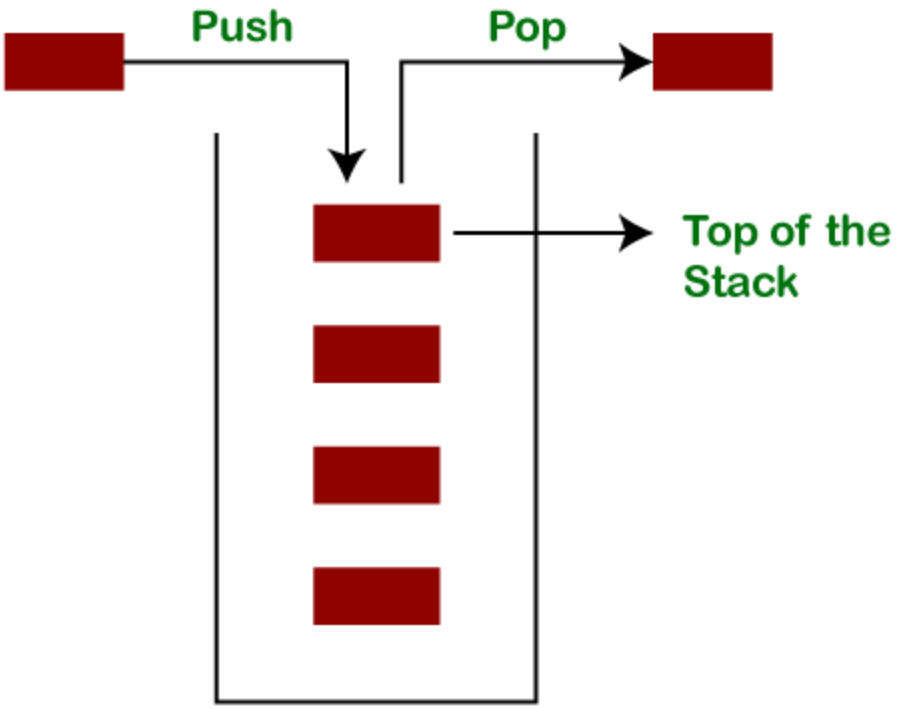
ut - Example (run) ×

```
run:
Ravi
Vijay
Ravi
Ajay
last:Ajay
BUILD SUCCESSFUL (total time: 0 seconds)
```

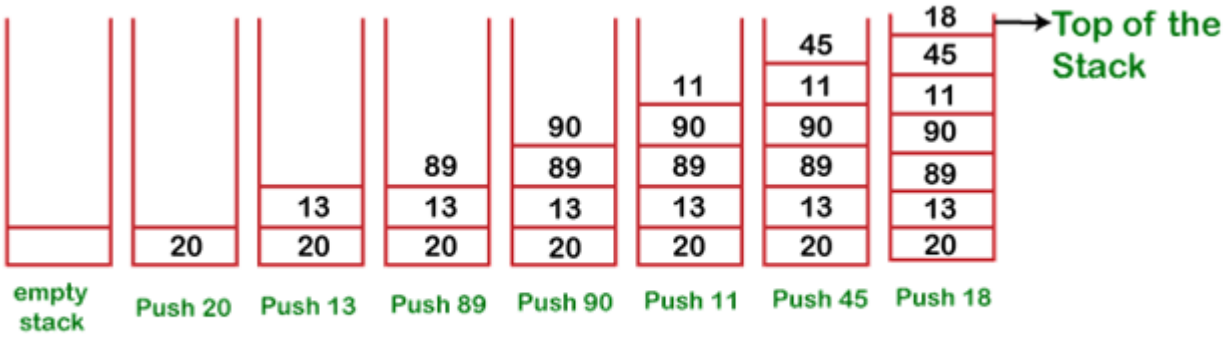
Stack


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

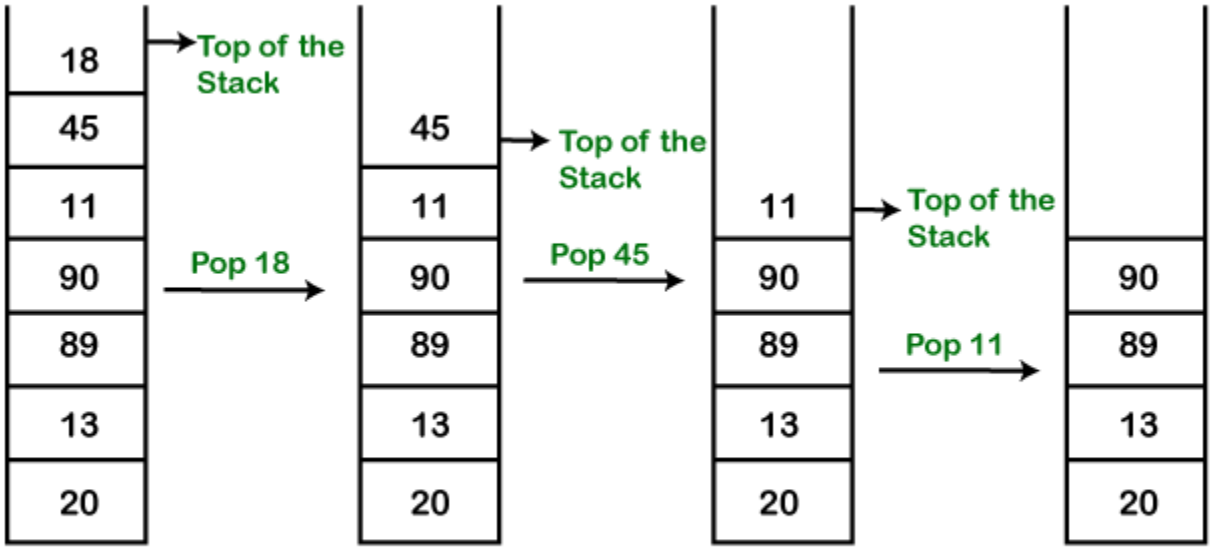
Stack



Stack



Push operation



Pop operation

Stack

- In the `Stack` class, the stack of objects results in a Last-In-First-Out (LIFO) behavior.
- It extends the `Vector` class to consider a vector as a stack.
- Stack only defines the default constructor that creates an empty stack.
- It includes all the methods of the vector class.
- This interface includes the following five methods:
  - `empty()`
  - `peek()`
  - `pop()`
  - `push(E item)`
  - `int search(Object o)`

Stack

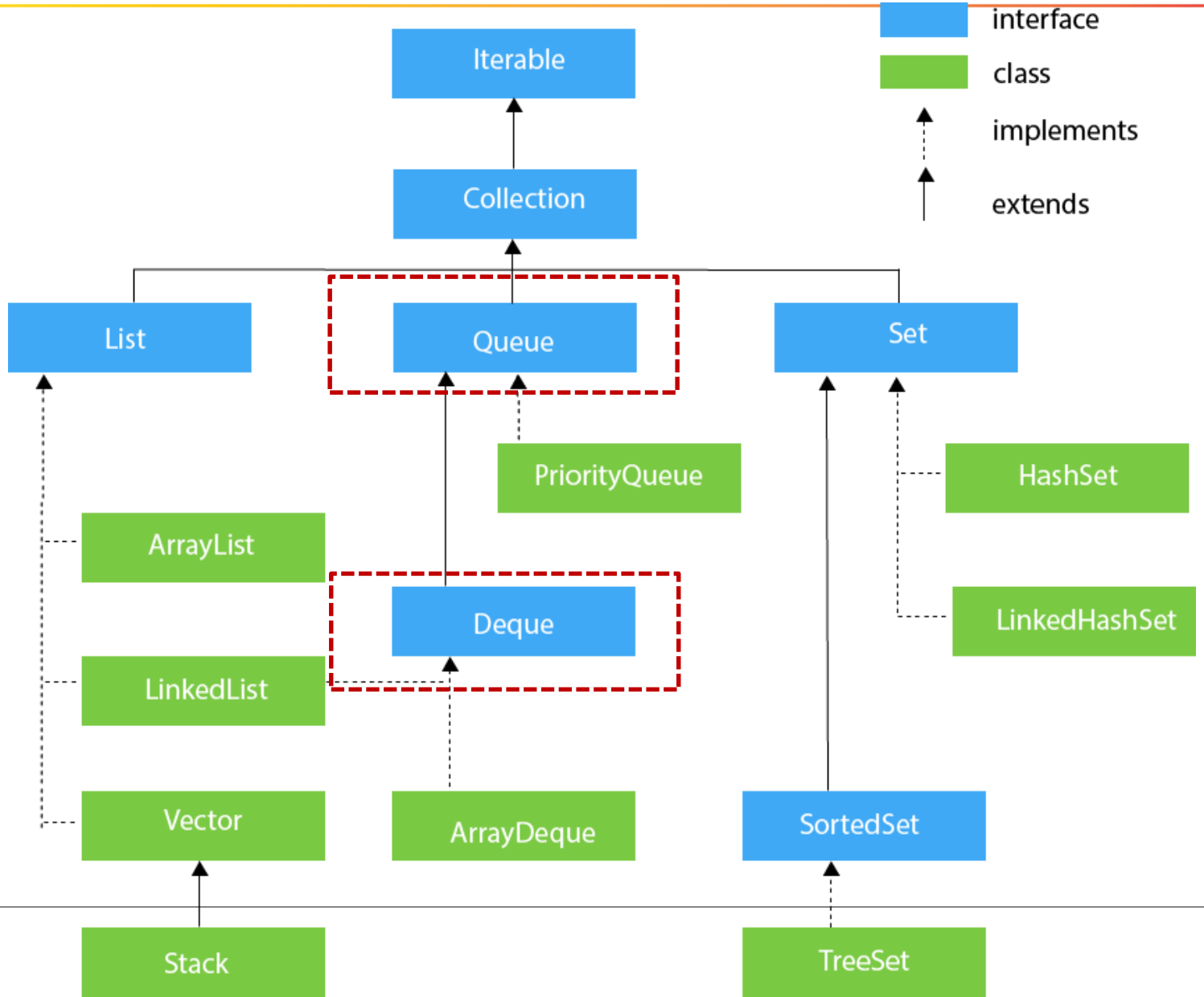
```java
import java.util.*;
class Example{
    public static void main(String args[]){
        //creating an instance of Stack class
        Stack<Integer> stk= new Stack<>();
        // checking stack is empty or not
        boolean result = stk.empty();
        System.out.println("Is the stack empty? " + result);
        // pushing elements into stack
        stk.push(78);
        stk.push(113);
        stk.push(90);
        stk.push(120);
        //prints elements of the stack
        System.out.println("Elements in Stack: " + stk);
        result = stk.empty();
        System.out.println("Is the stack empty? " + result);
    }
}
```

example.Example > main >

ut - Example (run) ×

```
run:
Is the stack empty? true
Elements in Stack: [78, 113, 90, 120]
Is the stack empty? false
BUILD SUCCESSFUL (total time: 0 seconds)
```

Queue & Deque Interface



Queue Interface

- A `Queue` is a collection for holding elements that needs to be processed.
- In `Queue`, the elements are normally ordered in First-In-First-Out (FIFO) manner.
- A queue can be arranged in other orders too.
- Every Queue implementation defines ordering properties.

Queue Interface

- In a FIFO queue, new elements are inserted at the end of the queue.
- LIFO queues or stacks order the elements in LIFO pattern.
- However, in any form of ordering, a call to the `poll()` method removes the head of the queue.

Deque

- A double ended queue is commonly called deque.
- It is a linear collection that supports insertion and removal of elements from both ends.
- Usually, `Deque` implementations have no restrictions on the number of elements to include.
- A `deque` when used as a queue results in FIFO behavior.
- The `Deque` interface and its implementations when used with the `Stack` class provides a consistent set of LIFO stack operations.
- The following Code Snippet displays `Deque`:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Methods of Deque

- `poll()`
- `peek()`
- `remove()`
- offer(E obj)
- element()

ArrayDeque

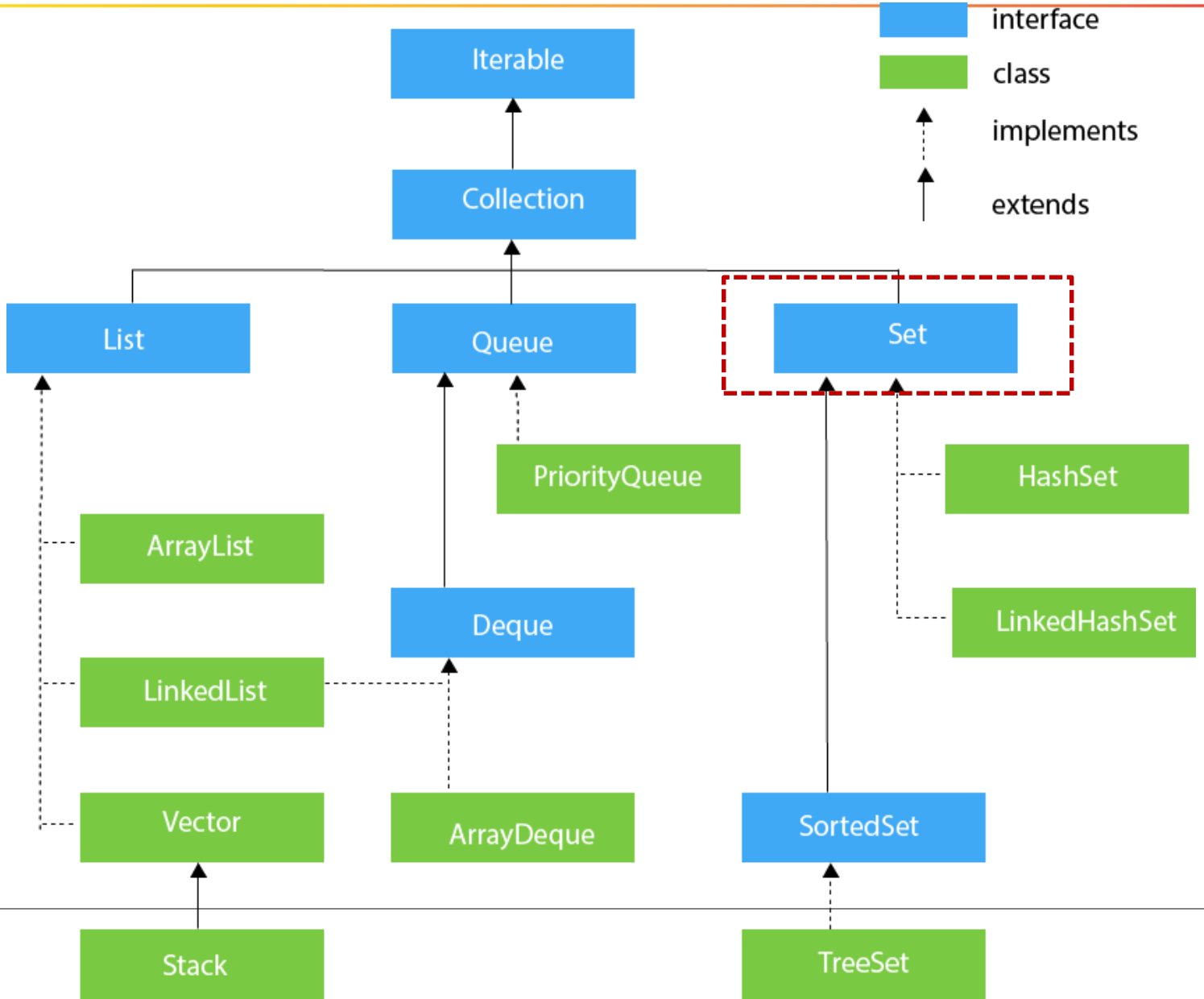
```
import java.util.*;
class Example{
    public static void main(String args[]){
        Deque<String> deque=new ArrayDeque<String>();
        deque.offer("arvind");
        deque.offer("vimal");
        deque.add("mukul");
        for(String s:deque){
            System.out.println(s);
        }
    }
}
```

example.Example

ut - Example (run)

```
run:
arvind
vimal
mukul
BUILD SUCCESSFUL (total time: 0 seconds)
```

Sets


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

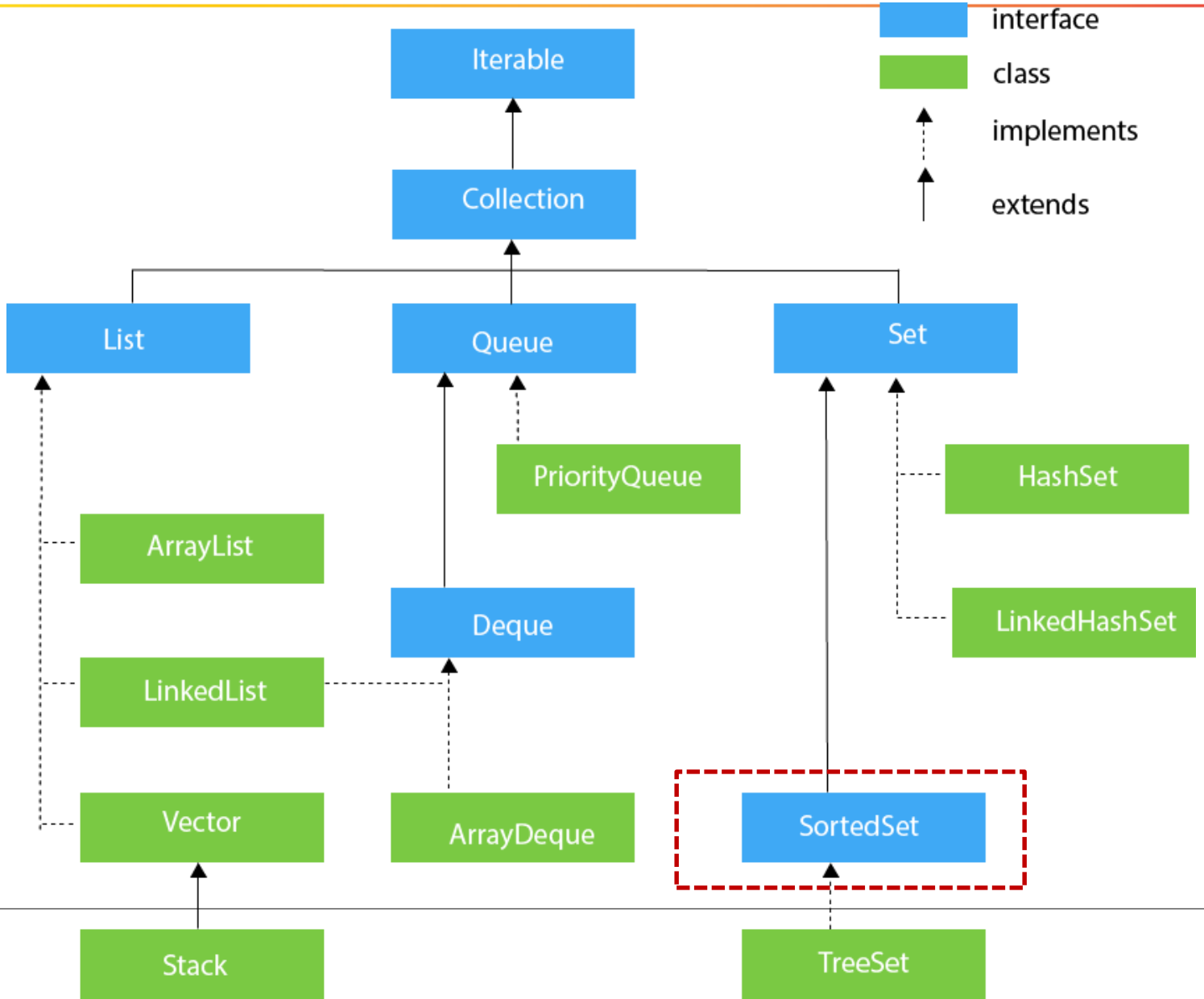
Sets

- The `Set` interface creates a list of unordered objects.
- It creates non-duplicate list of object references.
- The `Set` interface inherits all the methods from the `Collection` interface, except those methods that allow duplicate elements.
- The Java platform contains three general-purpose `Set` implementations. They are as follows:
  - `HashSet`
  - `TreeSet`
  - `Link`
- The `Set` interface is an extension of the `Collection` interface and defines a set of elements.
- The difference between `List` and `Set` is that, the `Set` does not permit duplication of elements.
- `Set` is used to create non-duplicate list of object references.
- Therefore, `add()` method returns false if duplicate elements are added.

Methods of Set Interface

- `containsAll(Collection<?> obj)`
- `addAll(Collection<? extends E> obj)`
- `retainAll(Collection<?> obj)`
- `removeAll(Collection<?> obj)`

SortedSet Interface



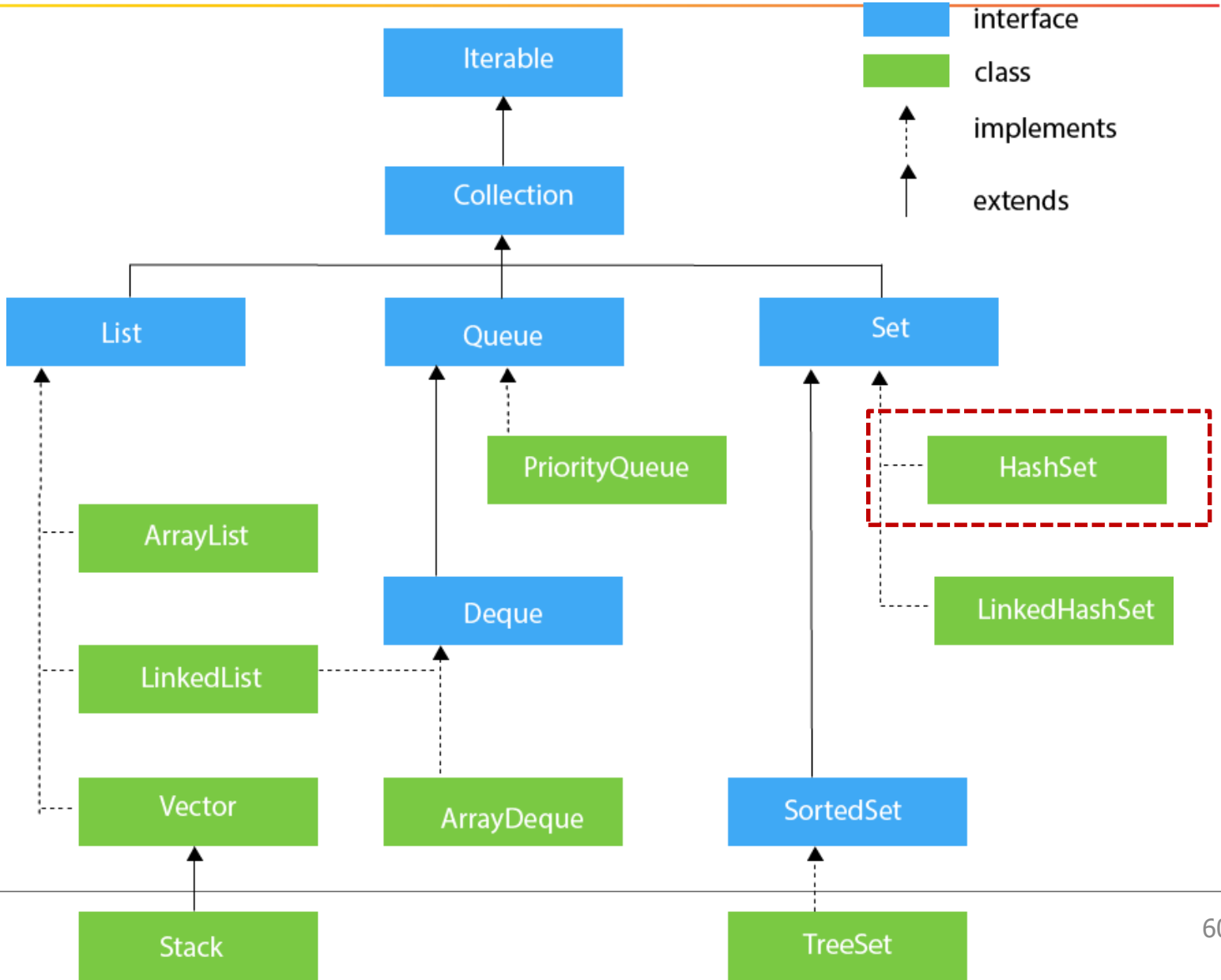
SortedSet Interface

- The `SortedSet` interface extends the `Set` interface and its iterator traverses its elements in the ascending order.
- Elements can be ordered by natural ordering, or by using a Comparator that a user can provide while creating a sorted set.
- `SortedSet` is used to create sorted lists of non-duplicate object references.
- The ordering of a sorted set should be consistent with `equals()` method.
- A sorted set performs all element comparisons using the `compareTo()` or `compare()` method.

Methods of SortedSet Interface

- `first()`
- `last()`
- `headSet (E endElement)`
- `subSet (E startElement, E endElement)`
- `tailSet (E fromElement)`

HashSet Class



HashSet Class

- HashSet class implements the Set interface and creates a collection that makes use of a hashtable for data storage.
- This HashSet class allows null element.
- The HashSet class provides constant time performance for the basic operations.
- The constructors of the HashSet class are as follows:
  - HashSet()
  - HashSet(Collection<? extends E> c)
  - HashSet(int size)
  - HashSet(int size, float fillRatio)

HashSet Class

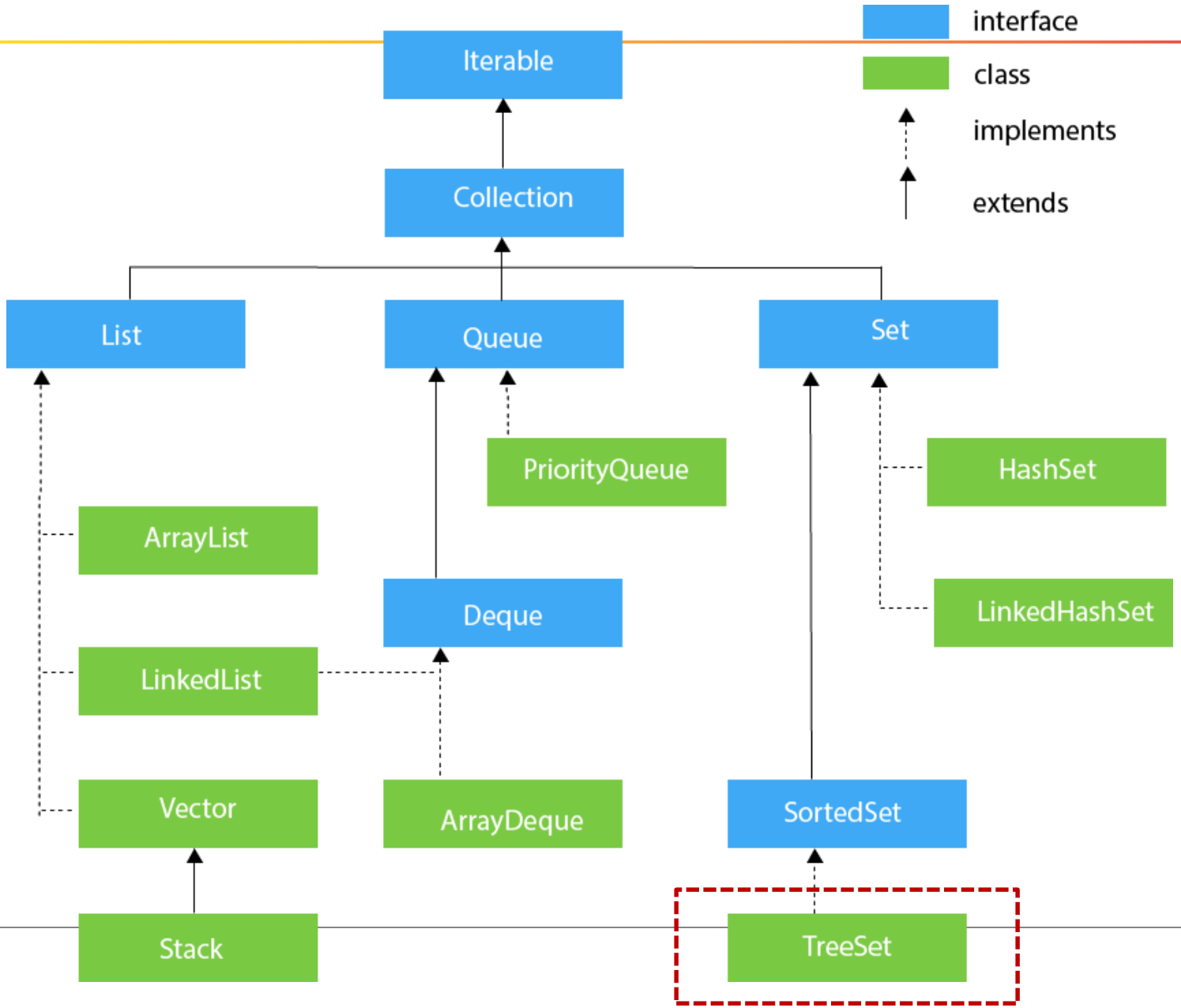
```
import java.util.*;
class Example{
    public static void main(String args[]){
        HashSet<String> set=new HashSet();
        set.add("One");
        set.add("Two");
        set.add("Three");
        set.add("Four");
        set.add("Five");
        Iterator<String> i=set.iterator();
       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

example.Example > main > set >

ut - Example (run) ×

```
run:
Five
One
Four
Two
Three
BUILD SUCCESSFUL (total time: 0 seconds)
```

TreeSet Class



TreeSet Class

- TreeSet class implements the NavigableSet interface and uses a tree structure for data storage.
- The elements can be ordered by natural ordering or by using a Comparator provided at the time of Set creation.
- Objects are stored in ascending order and therefore accessing and retrieving an object is much faster.
- TreeSet is used when elements needs to be extracted quickly from the collection in a sorted manner.
- This class includes the following constructors:
  - TreeSet()
  - TreeSet(Collection<? extends E> c)
  - TreeSet(Comparator<? super E> c)
  - TreeSet(SortedSet<E> s)

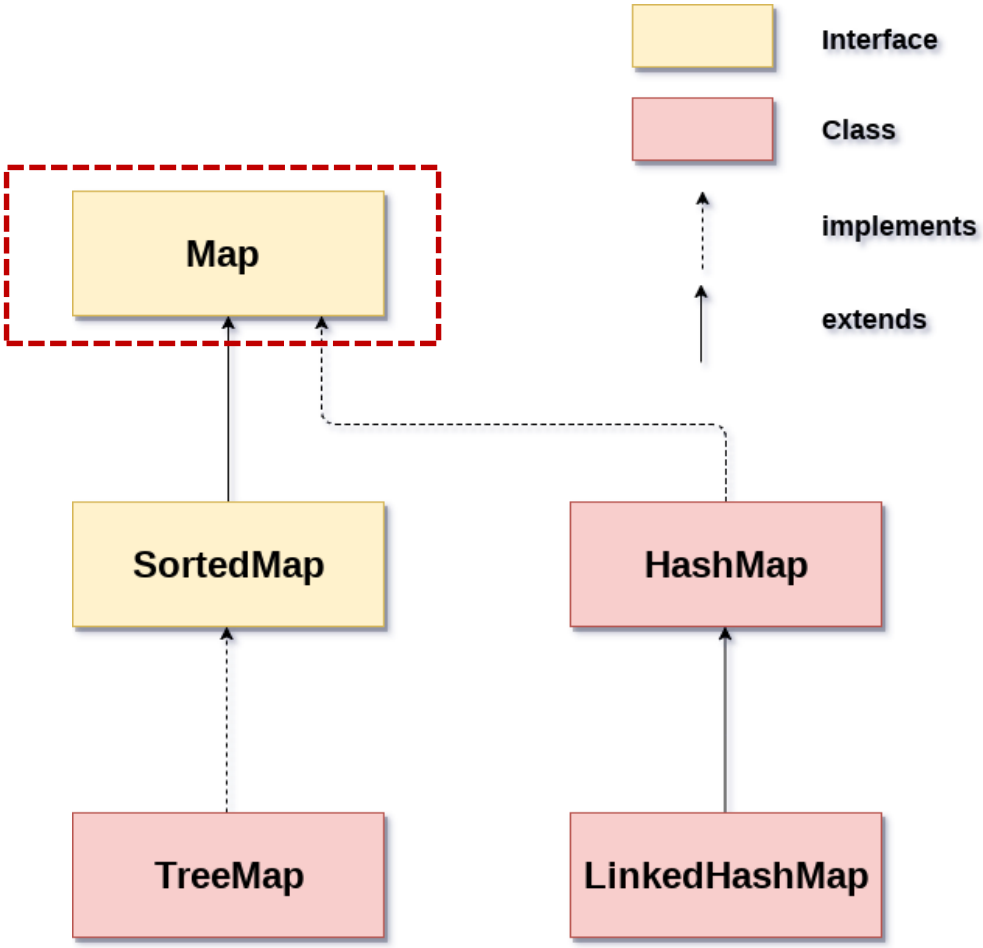
TreeSet Class

```
import java.util.*;
class Example{
    public static void main(String args[]){
        TreeSet<String> set=new TreeSet<String>();
        set.add("Ravi");
        set.add("Vijay");
        set.add("Ajay");
        System.out.println("Traversing element through Iterator in descending order");
        Iterator i=set.descendingIterator();
        while(i.hasNext())  {
            System.out.println(i.next());
        }
    }
}
```

ut - Example (run)

```
run:
Traversing element through Iterator in descending order
Vijay
Ravi
Ajay
BUILD SUCCESSFUL (total time: 0 seconds)
```

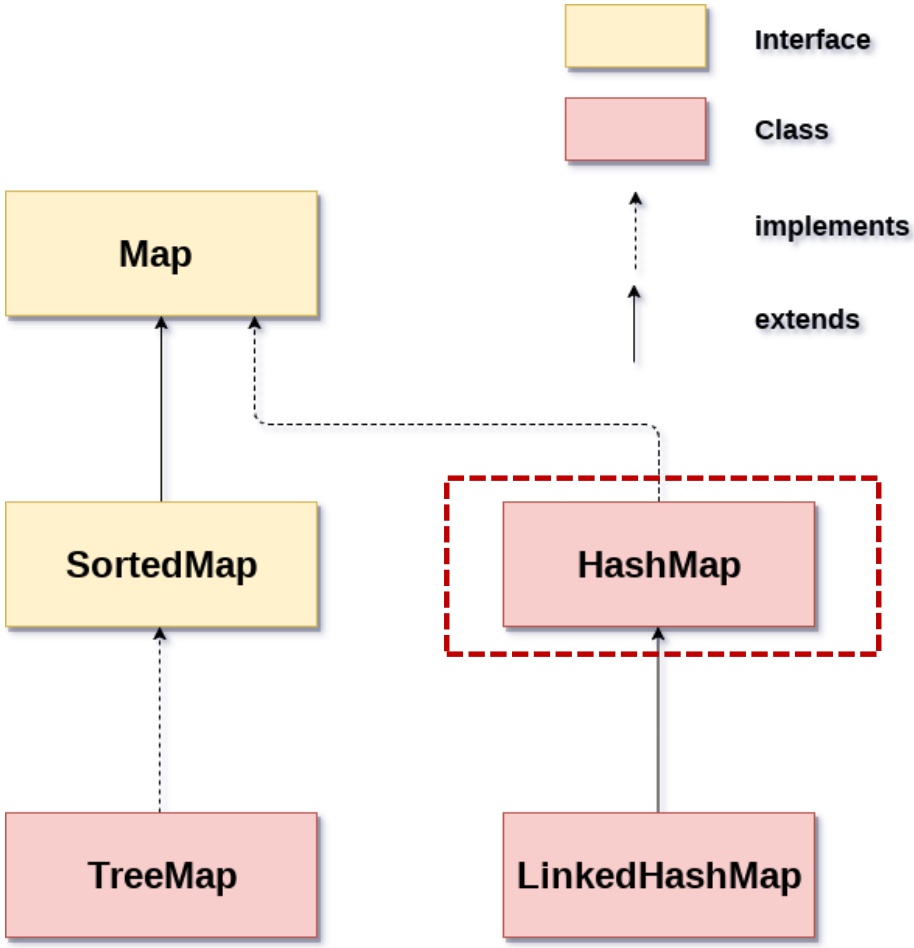
Maps


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

Maps

- A `Map` object stores data in the form of relationships between keys and values.
- Each key will map to at least a single value.
- If key information is known, its value can be retrieved from the `Map` object.
- Keys should be unique but values can be duplicated.
- The `Map` interface does not extend the `Collection` interface.
- The interface describes a mapping from keys to values, without duplicate keys.
- The Collections API provides three general-purpose `Map` implementations:
  - `HashMap`
  - `TreeMap`
  - `LinkedHashMap`
- The important methods of a `Map` interface are as follows:
  - `put(K key, V value)`
  - `get(Object key)`
  - `containsKey(Object key)`
  - `containsValue(Object value)`
  - `size()`
  - `values()`

HashMap Class



HashMap Class

- The `HashMap` class implements the `Map` interface and inherits all its methods.
- An instance of `HashMap` has two parameters: initial capacity and load factor.
- Initial capacity determines the number of objects that can be added to the `HashMap` at the time of the `Hashtable` creation.
- The load factor determines how full the Hashtable can get, before its capacity is automatically increased.
- The constructors of this class are as follows:
  - `HashMap()`
  - `HashMap(int initialCapacity)`
  - `HashMap(int initialCapacity, float loadFactor)`
  - `HashMap(Map<? extends K,? extends V> m)`

HashMap Class

```
import java.util.*;
class Example{
    public static void main(String args[]){
        HashMap<Integer,String> map=new HashMap<Integer,String>();
          map.put(1,"Mango");  //Put elements in Map
          map.put(2,"Apple");
          map.put(3,"Banana");
          map.put(4,"Grapes");

          System.out.println("Iterating Hashmap...");
          for(Map.Entry m : map.entrySet()){
           System.out.println(m.getKey()+" "+m.getValue());
          }
    }
}
```

```
ut - Example (run)
run:
Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes
BUILD SUCCESSFUL (total time: 0 seconds)
```

Hashtable Class

- The `Hashtable` class implements the `Map` interface but stores elements as a key/value pairs in the hashtable.
- While using a `Hashtable`, a key is specified to which a value is linked.
- The key is hashed and then the hash code is used as an index at which the value is stored.
- The class inherits all the methods of the `Map` interface.
- To retrieve and store objects from a hashtable successfully, the objects used as keys must implement the `hashCode()` and `equals()` method.
- The constructors of this class are as follows:
  - `Hashtable()`
  - `Hashtable(int initCap)`
  - `Hashtable(int intCap, float fillRatio)`
  - `Hashtable(Map<? extends K,? extends V> m)`

Hashtable Class

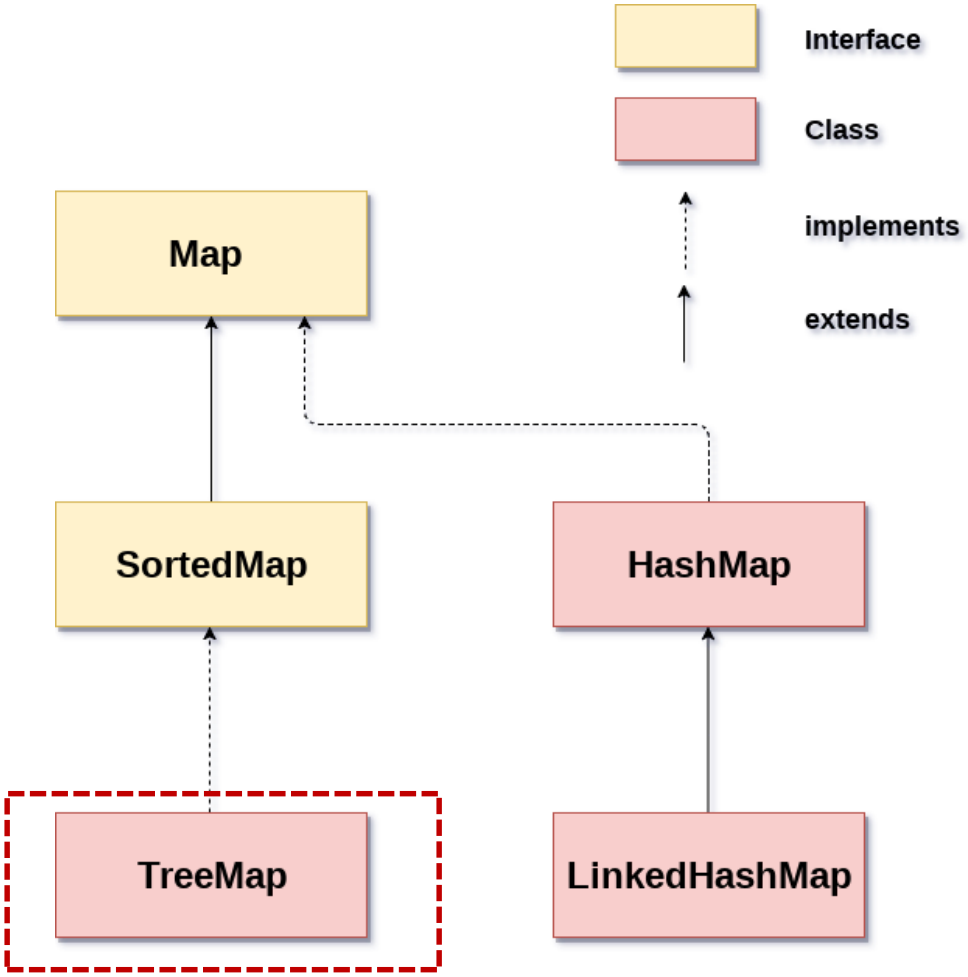
```
import java.util.*;
class Example{
    public static void main(String args[]){
        Hashtable<Integer,String> map=new Hashtable<Integer,String>();
            map.put(100,"Amit");
            map.put(102,"Ravi");
            map.put(101,"Vijay");
            map.put(103,"Rahul");
            System.out.println("Before remove: "+ map);
              // Remove value for key 102
              map.remove(102);
              System.out.println("After remove: "+ map);
    }
}
```

example.Example > main >

ut - Example (run) ×

```
run:
Before remove: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
After remove: {103=Rahul, 101=Vijay, 100=Amit}
BUILD SUCCESSFUL (total time: 0 seconds)
```

TreeMap Class



TreeMap Class

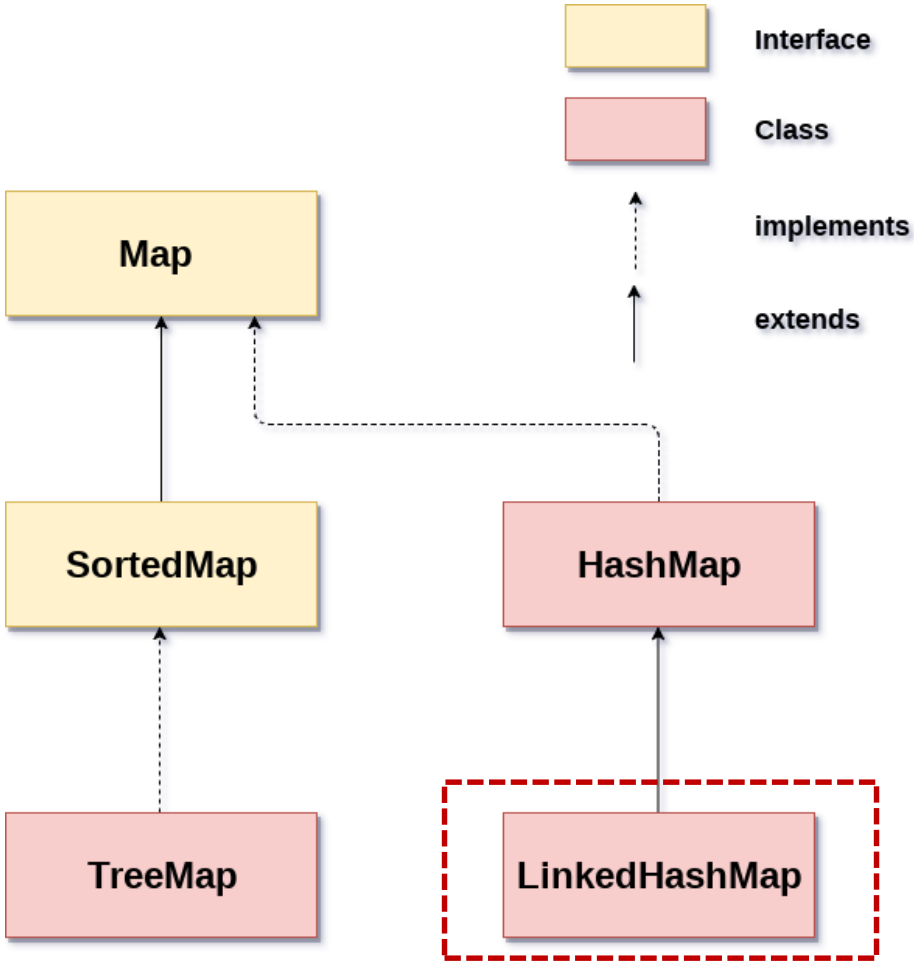
- The `TreeMap` class implements the `NavigableMap` interface but stores elements in a tree structure.
- The `TreeMap` returns keys in sorted order.
- If there is no need to retrieve `Map` elements sorted by key, then the `HashMap` would be a more practical structure to use.
- The constructors of this class are as follows:
  - `TreeMap()`
  - `TreeMap(Comparator<? super K> c)`
  - `TreeMap(Map<? extends K,? extends V> m)`
  - `TreeMap(SortedMap<K,? extends V> m)`
- The important methods of the `TreeMap` class are as follows:
  - `firstKey()`
  - `lastKey()`
  - `headMap(K toKey)`
  - `tailMap(K fromKey)`

TreeMap Class

```
import java.util.*;
class Example{
    public static void main(String args[]){
        TreeMap<Integer,String> map=new TreeMap<Integer,String>();
            map.put(100,"Amit");
            map.put(102,"Ravi");
            map.put(101,"Vijay");
            map.put(103,"Rahul");

            for(Map.Entry m:map.entrySet()){
             System.out.println(m.getKey()+" "+m.getValue());
            }
    }
}
```

```
ut - Example (run)
run:
100 Amit
101 Vijay
102 Ravi
103 Rahul
BUILD SUCCESSFUL (total time: 0 seconds)
```

LinkedHashMap Class



LinkedHashMap Class

- `LinkedHashMap` class implements the concept of hashtable and the linked list in the `Map` interface.
- A `LinkedHashMap` maintains the values in the order they were inserted, so that the key/values will be returned in the same order that they were added to this `Map`.
- The constructors of this class are as follows:
  - `LinkedHashMap()`
  - `LinkedHashMap(int initialCapacity)`
  - `LinkedHashMap(int initialCapacity, float loadFactor)`
  - `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`
  - `LinkedHashMap(Map<? extends K,? extends V> m)`
- The important methods in `LinkedHashMap` class are as follows:
  - `clear()`
  - `containsValue(Object value)`
  - `get(Object key)`
  - `removeEldestEntry(Map.Entry<K,V> eldest)`

LinkedHashMap Class

```
import java.util.*;
class Example{
    public static void main(String args[]){
        Map<Integer,String> map=new LinkedHashMap<Integer,String>();
         map.put(101,"Amit");
         map.put(102,"Vijay");
         map.put(103,"Rahul");
         System.out.println("Before invoking remove() method: "+map);
        map.remove(102);
        System.out.println("After invoking remove() method: "+map);
    }
}
```

ut - Example (run)

```
run:
Before invoking remove() method: {101=Amit, 102=Vijay, 103=Rahul}
After invoking remove() method: {101=Amit, 103=Rahul}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Summary

- The java.util package contains the definition of number of useful classes providing a broad range of functionality.
- The List interface is an extension of the Collection interface.
- The Set interface creates a list of **unordered** objects.
- A Map object stores data in the form of relationships between **keys and values**.
- A Queue is a collection for holding elements before processing.
- ArrayDeque class does not put any restriction on capacity and does not allow null values.

Student Activities

- Compare types of collections
- Source: (review slide65)
  - Allow user entry 5 number
  - Print ordered list

ĐẠI HỌC FPT CẦN THƠ