R.M.K Engineering College

Department of Computer Science and Engineering

22CS202 - Java Programming

Objective

- To explain object oriented programming concepts and fundamentals of java.
- To apply the principles of packages, interfaces and exceptions.
- To develop a java application with I/O streams, threads and generic programming.
- To build applications using strings and collections.
- To apply the JDBC concepts.

Unit IV –STRING HANDLING AND COLLECTIONS

Lambda Expressions - String Handling – Collections: The Collection Interfaces, The Collection Classes – Iterator – Map - Regular Expression Processing.

Lambda Expressions

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface.

Java Lambda Expression Syntax
(argument-list) -> { body }

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
```

One Parameter Syntax

```
(p1) -> {
//Body of single parameter lambda
}
```

Two Parameter Syntax

```
(p1,p2) -> {
//Body of multiple parameter lambda
}
```

Without Lambda Expression

```
interface Drawable{
   public void draw();
}
public class LambdaExpressionExample
{
   public static void main(String[] args)
{
      int width=10;

      //without lambda, Drawable implementation using anonymous class
      Drawable d=new Drawable()
      {
         public void draw()
          {
           System.out.println("Drawing "+width);
          }
      };
      d.draw();
   }
}
```

Output:

```
Drawing 10
```

with Lambda Expression

```
interface Drawable{
    public void draw();
}

public class LambdaExpressionEx {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->
        {
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Lambda Expression Example: No Parameter

```
interface Sayable{
    public String say();
}
public class LambdaExpressionEx
{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

Lambda Expression Example: Single Parameter

```
interface Sayable{
    public String say(String name);
}
public class LambdaExpressionEx{
    public static void main(String[] args)
{
        Sayable s1=(name)->
       {
            return "Hello, "+name;
        };
        System.out.println(s1.say("Sonoo"));
        // You can omit function parentheses
        Sayable s2= name ->
       {
            return "Hello, "+name;
        };
        System.out.println(s2.say("Sonoo"));
    }  }
```

Lambda Expression Example: Multiple Parameters

```
interface Addable{
    int add(int a,int b);
}
public class LambdaExpressionEx{
    public static void main(String[] args) {
        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

Lambda Expression Example: with or without return keyword

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample6 {
    public static void main(String[] args) {

        // Lambda expression without return keyword.
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Lambda expression with return keyword.
        Addable ad2=(int a,int b)->{
                            return (a+b);
                            };
        System.out.println(ad2.add(100,200));
    }  }
```

STRING

String is probably the most commonly used class in java library. String class is encapsulated under java.lang.package. In java, every string that you create is actually an object of type string. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.

What is an Immutable Object?

An object state cannot be changed after it is created is known as an immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper class’s objects are immutable

Strings and its Methods

- length()
- concat()
- indexOf()
- charAt()
- replace()
- tolowerCase()
- toUpperCase()
- compareTo()
- trim()
- substring()

Creating a String :

There are two ways to create a string in java:

- •String literal
- •Using new keyword

String literal:

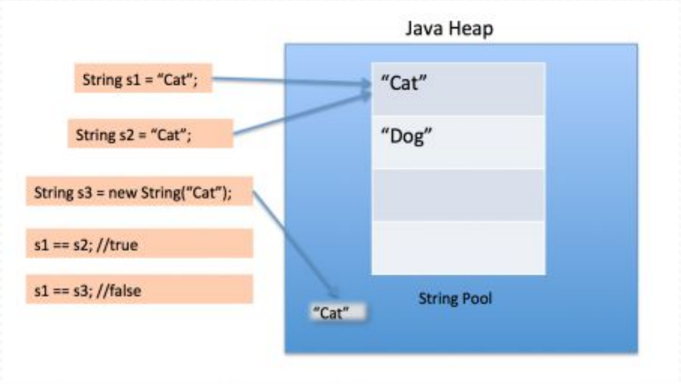
In java, Strings can be created like this: Assigning a string to a string instance:

String str=“Welcome”;

Using new keyword:

String str1=new String(“Welcome”)

```
class StringDemo1
{
 public static void main(String       args[])
{
        String s1=“Cat";
         String s2=“Cat";
        String s3=new String(“Cat");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);   }}
```



String length() method:

This method returns the length of the string. The length is equal to the number of 16-bit Unicode characters in the string.

Syntax: int length()

```
public class Example
{
public static void main(String args[])
{
String str1="";
String str2="Welcome";
System.out.println("Length of the first string is"+str1.length());
System.out.println("Length of the 2nd string is"+str2.length());
}}
```

Concatenating String :

There are two ways to concatenate two or more strings.

- Using concat() method
- Using + operator

Using concat() method:

```
public class Example
{
public static void main(String args[])
{
String s="Hi";
String str1="Welcome";
String str2=s.concat(str1);
String str3="Welcome".concat(" to Java");
System.out.println(str2);
System.out.println(str3);
}}
```

Using + operator:

```
public class Example
{
public static void main(String args[])
{
String str="Hi";
String str1="Welcome";
String str2=str+str1;
System.out.println(str2);
}
}
```

String indexOf() Method:

This method returns the index within this string of the first occurrence of the specified character or -1, if the character does not occur.

Syntax:

str.indexOf(ch);

```
public class Example
{
public static void main(String args[])
{
String str1="Welcome";
System.out.println(str1.indexOf("e"));}
}
```

String charAt() Method:

This method returns the character located at the string's specified index. The string indexes start from zero.

Syntax:

char charAt(int index);

```
public class Example
{
public static void main(String args[])
{
String str1="Welcome";
char ch1=str1.charAt(0);
System.out.println("Character at 0 index is:"+ch1);
}
}
```

String replace() Method:

This method returns a new string resulting from replacing all occurrences of oldChar in this string with newChar

Syntax:

String replace(char oldChar, char newChar);

```
public class Example
{
public static void main(String args[])
{
String str1="Welcome";
System.out.println(str1.replace('W','H'));
}
}
```

String toUpperCase() Method:

This method returns string with all lowercase character changed to uppercase.

Syntax:

String toUpperCase()

```
public class Example
{
public static void main(String args[])
{
String str1="Welcome";
System.out.println(str1.toUpperCase()
);
}
}
```

String toLowerCase() Method:

This method returns string with all lowercase character changed to uppercase.

Syntax:

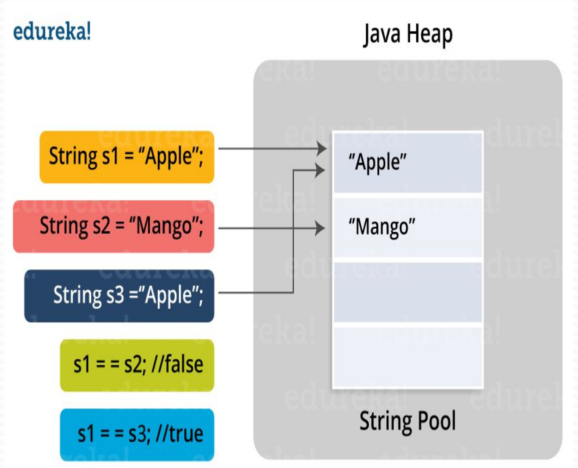
String toLowerCase()

```
public class Example
{
public static void main(String args[])
{
String str1="Welcome";
System.out.println(str1.toLowerCase());
}
}
```

String Comparison:

String comparison can be done in 3 ways.

- •Using equals() method
- •Using == operator
- •By CompareTo() method



Using == operator:

== operator compares two object reference to check whether they refer to same instance. This also, well return on successful match.

```
public class Example
{
public static void main(String args[]){
String s1="Apple";
String s3=new String("Apple");
System.out.println(s1==s3);     //false  }}
```

Using equals() method:

Equals() method compares two strings for equality.

Syntax: boolean equals(Object str)

```
public class Example
{
public static void main(String args[]){
String s1="Hello";
String s2=new String("Hello");
System.out.println(s1.equals(s2));    //true
}}
```

By compareTo() method:

compareTo() method compares values and returns an int which tells if the string compared Is less than, equal to or greater than other string.

Syntax:

int compareTo(String str)

```
class StringDemo1
{
public static void main(String args[])
{
String s1=“Abi";
String s2=“abi";
int str2=s1.compareTo(s2);
System.out.println(str2);
}}
```

```
s1==s2          //0
s1>s2      //+iv
s1<s2      //-iv
```

Output:

```
s1.compareTo(s2);  //-32 (value for A is 65
     – value for a is 97)
```

String compareToIgnoreCase() method:

This method compares two srings lexicographicaally, ignoring case differences.

Syntax: int compaerToIgnoreCase(String str)

```
class StringDemo1
{
public static void main(String args[])
{
String str="Hello";
String str1="hello";
int str2=str.compareToIgnoreCase(str1);
System.out.println(str2);
}
}
```

Output: 0

String trim() Method:

This method returns a copy of the string, with leading and trailing whitespace omitted.

Syntax:

String trim()

```
public class Example
{
public static void main(String args[])
{
String str=" Welcome ";
System.out.println(str.trim());
}
}
```

String substring(beginIndex, endIndex) Method:

This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or up to endIndex -1

Syntax:

String substring(int beginIndex, int endIndex)

```
public class Example
{
public static void main(String args[])
{
String str="Welcome";
System.out.println(str.substring(2,6));
}}
```

String Buffer Class

Java StringBuffer class is used to created mutable(modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Creating a String:

```
StringBuffer str=newStringBuffer(“Hello);
System.out.println(str);
    class StringDemo1
    {
    public static void main(String   args[])
    {
    StringBuffer str1=new       StringBuffer("Hello");
    System.out.println(str1);
    }}
```

String Buffer Class and its Methods

- append(String)
- insert(index,String)
- delete(start_index,end_index)
- reverse()
- replace(start_index,end_index,String)
- length()
- substring(start_index, end_start)
- indexOf(String)
- charAt(index)

StringBuffer append() Method:

The java.lang.StringBuffer.append(String str) method appends the specified string to this character sequence.

Syntax: StringBuffer append(String str)

```
public class Example

{     public static void main(String args[])

{

StringBuffer sb=new StringBuffer("Hello");

sb.append("Java");

System.out.println(sb);

}}
```

StringBuffer insert() Method:

The java.lang.StringBuffer.insert(int offset, String str) method inserts the specified string to this character sequence.

Syntax: StringBuffer insert(int offset, String str)

```
public class Example

{

public static void main(String args[])

{

StringBuffer sb=new StringBuffer("Hello");

sb.insert(2,"Java");

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

StringBuffer delete() Method:

The java.lang.StringBuffer.delete(int start, int end) method is used to delete characters in the the specified index.

Syntax: StringBuffer delete(int start, int end)

```
public class Example

{

public static void main(String args[]) {

StringBuffer sb=new StringBuffer("Hello");

sb.delete(1,3);

System.out.println(sb);

}}
```

StringBuffer reverse() Method:

The java.lang.StringBuffer.reverse() method is used to reverse the specified string.

Syntax: StringBuffer reverse()

```
public class Example
{   public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
StringBuffer sb1=new StringBuffer("12345");
sb.reverse();
sb1.reverse();
System.out.println(sb);
System.out.println(sb1);}}
```

StringBuffer replace() Method:

The java.lang.StringBuffer.replace() method replaces the character in a substring of this StringBuffer with character s in the specified String.

Syntax: StringBuffer replace(int start, int end, String str)

```
public class Example

{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.replace(1,3,"Hi");

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

StringBuffer length() method:

This method returns the length of the string. The length is equal to the number of 16-bit Unicode characters in the string.

Syntax: int length()

```
public class StringDemo2
{
public static void main(String args[])
{
StringBuffer str1=new StringBuffer(" ");
StringBuffer str2=new StringBuffer("Welcome");
System.out.println("Length of the first string is"+str1.length());
System.out.println("Length of the first string is"+str2.length());
}}
```

StringBuffer indexOf() Method:

This method returns the index within this string of the first occurrence of the specified character or -1, if the character does not occur.

Syntax:

str.indexOf(ch);

```
public class StringDemo2
{
public static void main(String args[])
{
StringBuffer str1=new StringBuffer("Welcome");
System.out.println(str1.indexOf("e"));
}
}
```

StringBuffer charAt() Method:

This method returns the character located at the string’s specified index. The string indexes start from zero.

Syntax:

char charAt(int index);

```
public class Example
{
public static void main(String args[])
{
StringBuffer str1=new StringBuffer("Welcome");
char ch1=str1.charAt(0);
System.out.println("Character at 0 index is:"+ch1);
}}
```

String substring(beginIndex, endIndex) Method:

This method has two variants and returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string or up to endIndex -1

Syntax:

StringBuffer substring(int beginIndex, int endIndex)

```
public class Example
{
public static void main(String args[])
{
StringBuffer str1=new StringBuffer("Welcome");
System.out.println(str1.substring(2,6));
}}
```

Collection

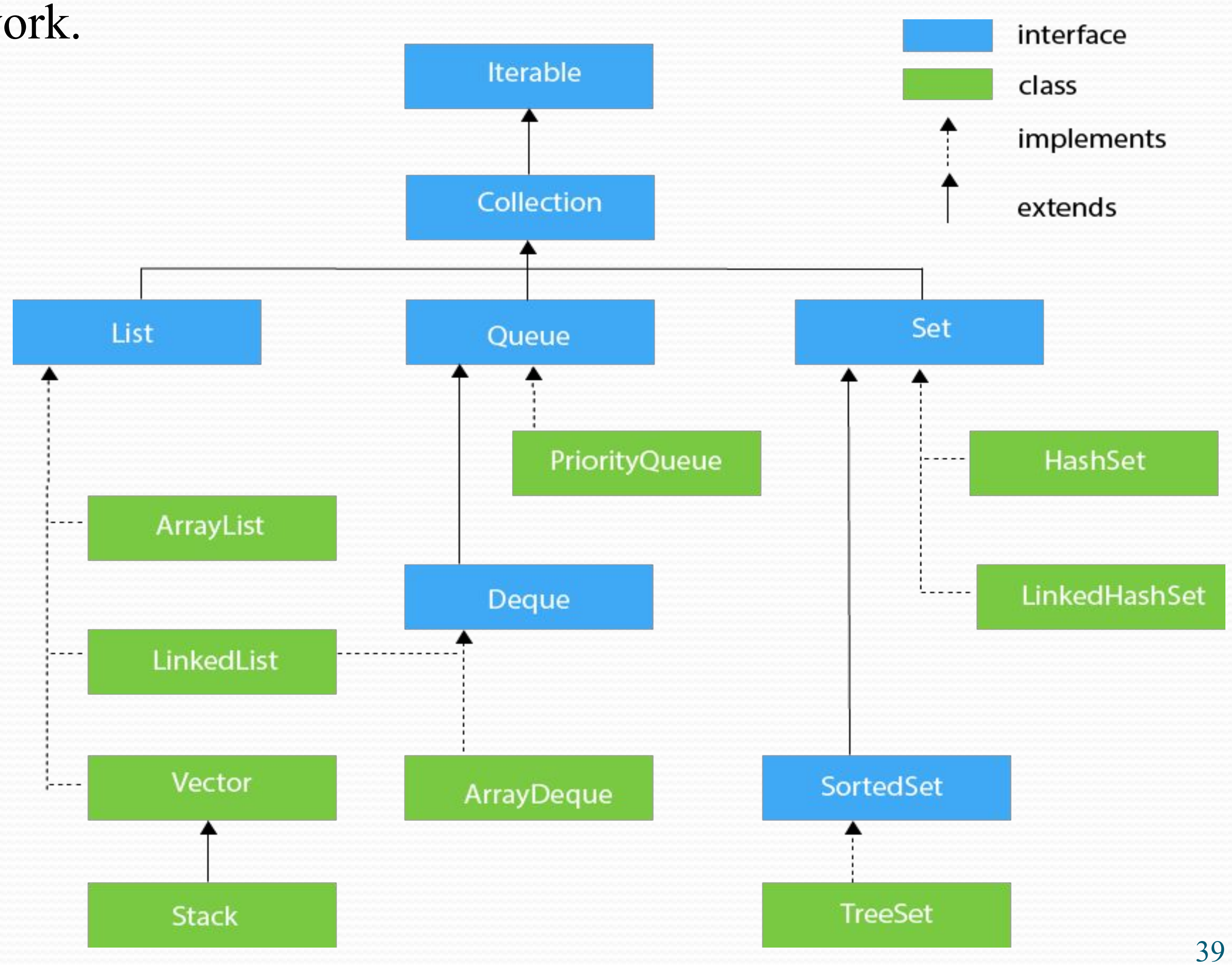
The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

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

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

Hierarchy of Collection Framework

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



Iterator interface

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

Methods of Iterator interface

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.

Iterable 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.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework.

List Interface

List interface is the child interface of Collection interface. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

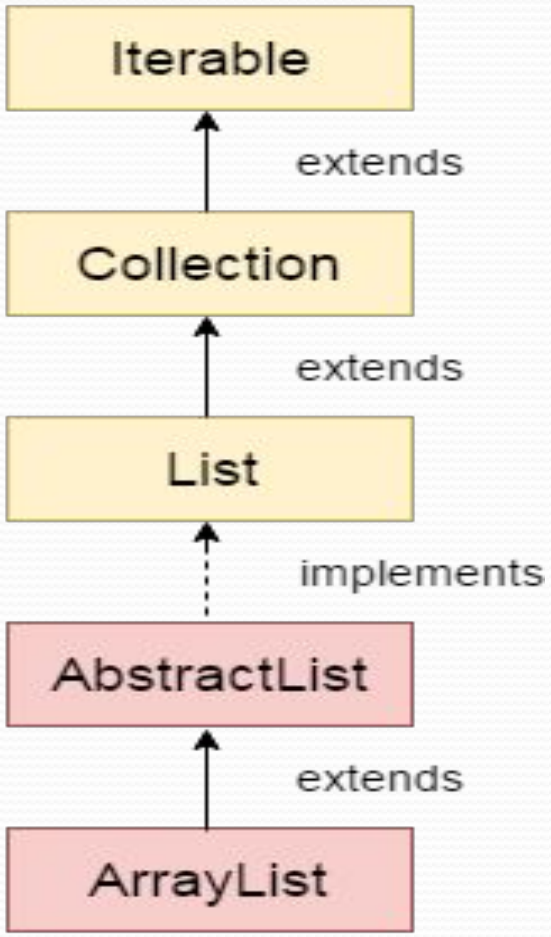
ArrayList

Java ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime.

```
import java.util.*;
class TestJavaCollection1
{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");    // System.out.println(list);
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}  }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```



LinkedList

This class is an implementation of the LinkedList data structure, where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part.

```
import java.util.*;
public class TestJavaCollection2{
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());
} } }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Vector

Vector uses a dynamic array to store the data elements.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Asha");
v.add("Anu");
v.add("Ravi");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Asha
Anu
Ravi
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(),boolean push(object o), which defines its properties.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Asha");
stack.push("Ajay");
stack.push("Amit");
stack.push("Ravi");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

```
Asha
Ajay
Amit
```

Set Interface

Set interface is the child interface of Collection interface. It does’t allow duplicate values.

Set interface is implemented by the classes HashSet, LinkedHashSet.

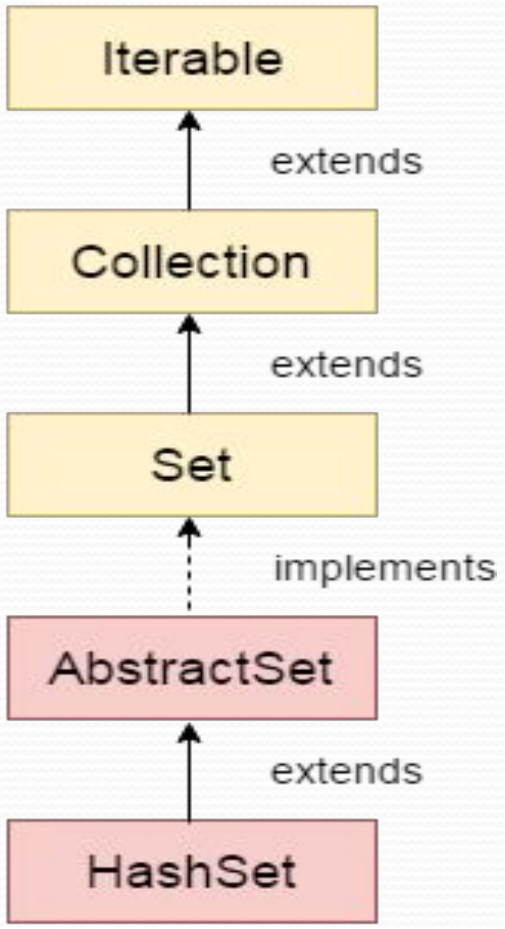
HashSet

HashSet stores the elements by using a mechanism called hashing. HashSet contains unique elements only. It doesn't follow any order to store the elements. HashSet allows null value.

```
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("One");
           set.add("Three");
           set.add("Four");
           set.add("Five");
           Iterator<String> i=set.iterator();
           while(i.hasNext())
           {
           System.out.println(i.next());
           }  }  }
```

Output:

```
Five
One
Four
Two
```



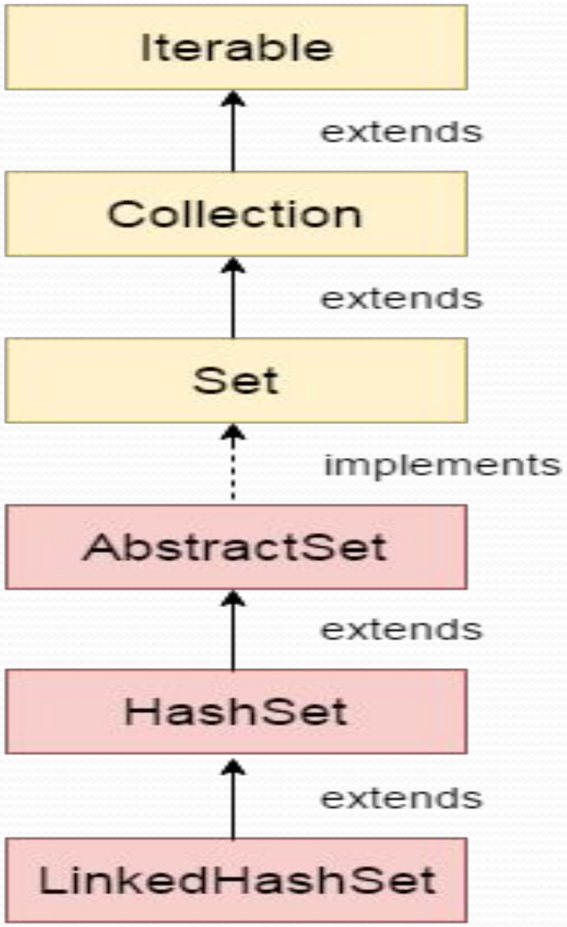
LinkedHashSet

Java LinkedHashSet class contains unique elements only like HashSet. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted.

```
import java.util.*;
class LinkedHashSet2{
 public static void main(String args[]){
  LinkedHashSet<String> al=new
LinkedHashSet<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());
  }
}  }
```

Output:

```
Ravi
Vijay
Ajay
```



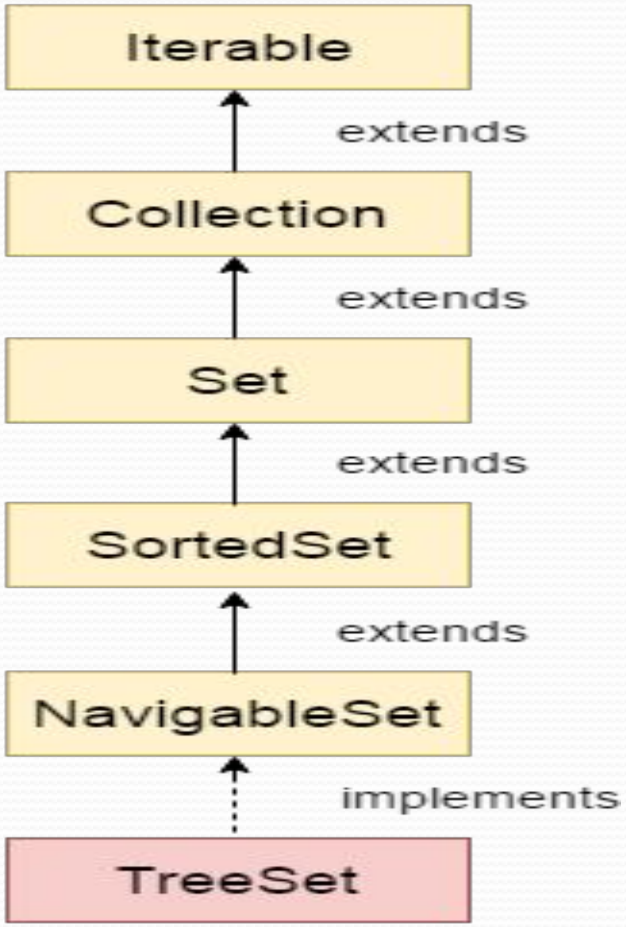
TreeSet

Java TreeSet class contains unique elements only like HashSet. Java TreeSet class maintains ascending order. Java TreeSet class doesn't allow null elements.

```
import java.util.*;
class TreeSet1{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }  }  }
```

Output:

```
Ajay
Ravi
Vijay
```



Queue Interface

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

PriorityQueue

```
import java.util.*;
class TestCollection{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit");
queue.add("Vijay");
queue.add("Karan");
System.out.println("head:"+queue.element());
System.out.println(queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} } }
```

Output:

```
head:Amit
head:Amit
iterating the queue elements:
Amit
Vijay
Karan
```

Deque

The Deque supports the addition as well as the removal of elements from both ends of the data structure. Therefore, a deque can be used as a stack or a queue. We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue. As a deque supports both, either of the mentioned operations can be performed on it. Deque is an acronym for "double ended queue".

ArrayDeque

```
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");
  //deque.remove();
  //Traversing elements
  for (String str : deque) {
  System.out.println(str);
  }
  }
}
```

```
Ravi
Vijay
Ajay
```

Collection class

```
import java.util.*;
public class CollectionsExample {
    public static void main(String a[]){
        List<String> list = new ArrayList<String>();
        list.add("C");
        list.add("Core Java");
        list.add("Advance Java");
        System.out.println(list);
        Collections.addAll(list, "Servlet","JSP");
        System.out.println("After adding elements collection value:"+list);
    }
}
```

Output:

```
[C, Core Java, Advance Java]
After adding elements collection value:
[C, Core Java, Advance Java, Servlet, JSP]
```

```
import java.util.*;
public class CollectionsExample {
    public static void main(String a[]){
        List<Integer> list = new ArrayList<Integer>();
        list.add(46);
        list.add(67);
        list.add(24);
        list.add(16);
        list.add(8);
        list.add(12);
        System.out.println("Value of maximum element from the
collection:" +Collections.max(list));
    }
}
```

iterator() Method

The iterator() method of Java Collection Interface returns an iterator over the elements in this collection.

```
import java.util.Collection;
import java.util.Iterator;
import java.util.concurrent.ConcurrentLinkedQueue;
public class JavaCollectionIteratorEx
{
   static int i = 1;
   public static void main(String[] args) {
        Collection<String> collection = new ConcurrentLinkedQueue<String>();
       collection.add("Ram");
       collection.add("Sham");
       collection.add("Mira");
       collection.add("Rajesh");
       Iterator<String> iterator = collection.iterator();  //Returns an iterator over the
elements
       while (iterator.hasNext()) {
          System.out.println(i++ + "." + iterator.next());
       }  }  }
```

Output:

```
1.Ram
2.Sham
3.Mira
4.Rajesh
```

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.

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

A map entry (key-value pair). The Map.entrySet method returns a collection-view of the map, whose elements are of this class.

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.

HashMap

```
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();
//Creating HashMap
   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());
   }
}
}
```

Output:

```
Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes
```

LinkedHashMap

```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){

  LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>();

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");


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

Output:

```
100 Amit
101 Vijay
102 Rahul
```

TreeMap

```
import java.util.*;
class TreeMap1{
 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());
     }
 }
}
```

output:

```
100 Amit
101 Vijay
102 Ravi
103 Rahul
```

Regular Expression

The Java Regex or Regular Expression is an API to define a pattern for searching or manipulating strings. **java.util.regex package.**

It is widely used to define the constraint on strings such as password and email validation.

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

Pattern Class - Defines a pattern (to be used in a search)
Matcher Class - Used to search for the pattern
PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

Matcher class

It implements the MatchResult interface. It is a regex engine which is used to perform match operations on a character sequence. The matcher() method is used to search for the pattern in a string. It returns a Matcher object which contains information about the search that was performed.

Pattern class

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine. the pattern is created using the Pattern.compile() method.

Expression	Description
[abc]	Find one character from the options between the brackets
[^abc]	Find one character NOT between the brackets
[0-9]	Find one character from the range 0 to 9

S. No.	Class/Interface	Description
1	Pattern Class	Used for defining patterns
2	Matcher Class	Used for performing match operations on text using patterns
3	PatternSyntaxException Class	Used for indicating syntax error in a regular expression pattern
4	MatchResult Interface	Used for representing the result of a match operation

```
import java.util.regex.*;
class RegexEx
{
public static void main(String args[]){
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));
//false (more than 6 char)
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false
($ is not matched)
}}
```

```
import java.io.*;
import java.util.regex.*;
 class RegEx {
        public static void main(String[] args)
     {
         // Checking all the strings using regex
         System.out.println(Pattern.matches("[b-z]?", "a"));
         // Check if all the elements are in range a to z
         // or A to Z
         System.out.println(
             Pattern.matches("[a-zA-Z]+", "GfgTestCase"));
         // Check if elements is not in range a to z
         System.out.println(Pattern.matches("[^a-z]?", "g"));
             }
}
```

Output:

```
false
true
false
```

```
import java.util.regex.*;
class RegEx {
    public static void main(String args[])
    {
        // Create a pattern to be searched
        // Custom pattern
        Pattern pattern = Pattern.compile("goo");
        // Search above pattern in "google.com"
        Matcher m = pattern.matcher("google.com");
        // Finding string using find() method
        while (m.find())
            // Print starting and ending indexes
            // of the pattern in the text
            // using this functionality of this class
            System.out.println("Pattern found from "+ m.start() + " to "+ (m.end() - 1));
    }}
```

Output: Pattern found from 0 to 2

```
import java.io.*;
import java.util.regex.*;
class RegExp
{
    public static void main(String[] args)
    {
        // Check if all elements are numbers
        System.out.println(Pattern.matches("\\d+", "1234"));

        // Check if all elements are non-numbers
        System.out.println(Pattern.matches("\\D+", "1234"));

        // Check if all the elements are non-numbers
        System.out.println(Pattern.matches("\\D+", "Gfg"));

        // Check if all the elements are non-spaces
        System.out.println(Pattern.matches("\\S+", "gfg"));
    }
}
```

Output:

```
true
false
true
true
```

Thank you!