Java Programming Beginners Course

PCW Courses Ltd
t/a PCWorkshops
Golden Cross House, 8 Duncannon Street, WC2N 5BY
Telephone: 020 7164 6359
Email: training@pcworkshopslondon.co.uk
PCW Courses Ltd. Trading As PCWorkshops
Company Number: 09990788

Java Programming Beginners Course

Data Structures

Table of Contents

Java Language Keywords

- Here is a list of keywords in the Java programming language.
- You cannot use any of the following as identifiers in your programs.
- The keywords const and goto are reserved, even though they are not currently used.
- true, false, and null might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

abstract
assert***
boolean
break
byte
case
catch
char
class
const*
continue
default
do
double
else
enum****
extends
final
finally
float
for
goto*
if
implements
import
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
strictfp**
super
switch
synchronized
this
throw
throws
transient
try
void
volatile
while

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

Week 3
Arrays

Declaring arrays:

How to declare or declare and initialise One Dimensional Arrays:

datatype[] arrayname;

arrayname = new datatype [number];

datatype[] arrayname = new datatype [number];

dataype[] arrayname= {value1, value2…};

Examples of declaring an array

```
String[] movies;
movies = new String[7];
OR
one can also use:    String[] movies = new String[7];
OR
String[] movies = {" ","Batman","Rambo","Scarface", “Cinderella”, “Schrek”, “ ”};
```

Looping through a one-dimensional movies array:

```
for ( int index=0; index  <  movies.length  ;  index++  ){
   System.out.println("The movie is " + movies [index] );
}
System.out.println();
```

Looping through a day- array:

```
String[] days = {"Mon","Tue","Wed","Thu", “Fri”, “Sat”, “Sun”};
for (int index= 0 ; index  < days.length ; index++  ){
   System.out.print("The day is " + days [index] + “\t”);
}
System.out.println();
```

Print all the arrays

```
System.out.println(“Movie program”);
for (int index= 0; index < prices.length  ; index++  ){
   System.out.print(days[index] + “\t” + prices [index] + “\t” + movies[index]);
}
System.out.println();
```

Exercise– single array: Retrieving values per index

```
String[] movies = {"None","Batman","Ramco","Scarface", “Cinderella”, “Schrek”, “None”};
String[] days = {"Mon","Tue","Wed","Thu", “Fri”, “Sat”, “Sun”};
double [] prices = {0.0 ,12.0,15.0, 12.0, 15.0, 18.0, 0.0};
double singleTicketPrice = 0.0;
String movieName = “”;
```

1. Get from the screen:
    a. the day of the week
    b. number of adults ,
    c. children (they pay 50%) and
    d. pensioners (they pay 30% of the adult price).
2. They have 200Pounds .

3. Loop the array to find the day they prefer and to find the row index .
4. Determine the single ticket price and the movieName
5. Calculate the total ticket price

6. Display:
    a. the day,
    b. the movieName,
    c. the single ticket price and
    d. If the full Price exceed £200,
        i. then print an appropriate message,
        ii. otherwise price the Price

Solution:

```
import java.util.Scanner;

public class ex1 {

	private static class innerClass {
		// variables
		String[] movies
		= {"None","Batman","Ramco","Scarface", "Cinderella", "Schrek", "None"};
		String[] days = {"Mon","Tue","Wed","Thu", "Fri", "Sat", "Sun"};
		double [] prices = {0.0 ,12.0,15.0, 12.0, 15.0, 18.0, 0.0};

		//scanners
		Scanner s = new Scanner(System.in);

		//methods
		String askForValue (String msg ){
			System.out.println(msg);
			String a = s.nextLine();
			return a;
		}
		int findMovie (String dOfW){
			int j = -1;
			for (int i = 0; i<days.length; i++){
				if (days[i].equalsIgnoreCase(dOfW)){
					j=i;
				}

			}
			return j;
		}
		double calcMovie (int arrayIndex, int a, int c, int p){
			double price = prices[arrayIndex];
			price = price * a + ( price*c/2) + (price * p * 0.3 );
			return price;
		}
		void printPrice (int arrayIndex, double price){
			System.out.println("Movie Name: "+ movies[arrayIndex]);
			System.out.println("Day: "+ days[arrayIndex]);
			System.out.println("Singe Ticket Price: "+ prices[arrayIndex]);
			if (price > 200){
				System.out.println("Price: "+ price + "It's over £200");
			} else{
				System.out.println("Full Price: "+ prices[arrayIndex]);
			}
		}
	}

	public static void main(String[] args) {
		innerClass i = new innerClass();
		String dayOfW = i.askForValue("Day of the week?");
		int adults = Integer.parseInt(i.askForValue("Adults?"));
		int children = Integer.parseInt(i.askForValue("Children?"));
		int pensioners = Integer.parseInt(i.askForValue("Pensioners?"));

		int arrayIndex = i.findMovie(dayOfW);
		double fullPrice = i.calcMovie(arrayIndex, adults, children, pensioners);
		i.printPrice(arrayIndex, fullPrice);

	}

}
```

Array Exercise

/* Description:
You have an array of salary values
It is year-end and you want to increase all the salaries with 7%
For example, a salary is 20000, then after the increase the salary is 20000 x 1.07 = 21400
Write a program that will increment all the salaries in the array with 7% and update the array with the new values. Then display all the new values.
*/

```
import java.util.Scanner;
class page8{ //class
public static void main (String[ ] args) { //main

    int i =0;
    double[] salaries= {20000,10000,24000};

    //amend all the salaries
    for(i=0;i< salaries.length; i++ ) {
                    salaries[i]= salaries [i]* 1.07;
    }

    //print all the salaries
    for(i=0;i< salaries.length; i++ ) {
            System.out.print(salaries [i]+"\t");
    }


} //main
} //class
```

Array Exercise

```
// Build array ccc from the values in array aaa, where the values in array aaa is NOT = 5.
// Display how many values NOT equals to 5  were found and transferred to array  ccc
// Display all values in array ccc
    //int aaa[] = {5,3,5,4,-5,5,6,2};
    //int ccc[] = {0,0,0,0,0,0,0,0};
```

```
import java.util.Scanner;
class page9{ //class
public static void main (String[ ] args) { //main

    int i =0;
    int x=0;
    int y=0;
    int aaa[] = {5,3,5,4,-5,5,6,2};
    int ccc[] = {0,0,0,0,0,0,0,0};


    for(i=0;i<aaa.length;i++ )
    {
            if (aaa[i]!=5) {
                    x++;
                    ccc[i]=aaa[i];
            }
    }//for


    for(i=0;i<ccc.length;i++ )
    {
            System.out.print(ccc[i]+"\t");
    }//for
            System.out.println();
            System.out.println("The  number  of replacements is  " + x);


} //main
} //class
```

Array Exercise

```
/*
You have 2 arrays: array arrayOne and array arrayTwo, assuming their length is the  same:
Create an array called arrayThree that contains the component wise sum of two arrays
e.g., if the input arrays are {0, 1, 2} and {2, 2, 3}
then the output is {0 + 2, 1 + 2, 2 +  3},
i.e., {2, 3, 5}.
If the input arrays have different numbers of elements,  no values are transferred and all the values in the new array are  0 .
int arrayOne[] = {0,1,2};
    int arrayTwo[]={2,2,3};
    int arrayThree[] = {0,0,0};

*/
```

```
import java.util.Scanner;
class page10{ //class
public static void main (String[ ] args) { //main

    int i =0;
    int x=0;
    int y=0;
    int arrayOne[] = {0,1,2};
    int arrayTwo[]= {2,2,3};
    int arrayThree[] = {0,0,0};

    x = arrayOne.length;
    y = arrayTwo.length;

    if (x==y){
        for (i=0; i<arrayOne.length; i++ ) {
            arrayThree[i]=arrayOne[i]+arrayTwo[i];
        }//for
    }//if

     for (i=0; i<arrayThree.length; i++ ) {
        System.out.println(arrayThree[i]+"\t");
    }//for


} //main
} //class
```

Array Exercise

/*
You have an double array named **oldArray** and initialised to oldArray{1.8, 3.6, 5.0, 2.0}.
Create an array named **newArray**:
For every entry in array **oldArray**, divide the entry by the next one, i.e. 1.8/3.6 etc.
Place the result of this division into the array **newArray**.
However, the last entry does not change and is simply transferred as is to the **newArray** array.
Hint: The resulting array: new = {0.5, 0.72, 2.5, 2.0}
Print the array **newArray**
*/

```
import java.util.Scanner;
class page11{ //class
public static void main (String[ ] args) { //main

    int index =0;
    double oldArray[] = {1.8,3.6,5.0,2.0};
    double newArray[] = {0.0,0.0,0.0,0.0};


    //build the array
    for(index=0;index<(oldArray.length-1);index++ )
    {
            newArray[index]= oldArray[index]/oldArray[index+1];
    }

    //transfer the last element
    newArray[oldArray.length-1]= oldArray[oldArray.length-1];

    //display the array
    for(index=0;index<newArray.length;index++ )
    {
            System.out.print(newArray[index]+"\t");
    }


} //main
} //class
```

2-dimensional Arrays

```
double [][] numbers = {
        {0.0 , 12.00 ,15.00 },
        {0.0 , 6.00 , 12,00  },
        {0.0 , 4.00 ,5.00 },
        };
```

```
double[ ][ ] prices = new double[3][3];
```

```
Double [ ][ ] number = new double[3][7];
```

OR:

```
int rows =0;
double [][] numbers = {
        {0.0 ,   12.00 , 15.00 , 12.00 , 15.00 , 18.00 ,  0.0},
        {0.0 ,    6.00 ,  7.50 ,   6.00 ,   7.50 ,    9.00 ,  0.0},
        {0.0 ,   4.00 ,   5.00 ,   4.00 ,    5.0 0,  6.00 ,   0.0},
        } ;
```

```
Arrayname.length                    // number of rows
Arrayname[index_of_row].length      // number of columns
```

```
String [ ][ ] movies = {
            {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"},
            {"None","Schrek","Batman","Rocky","Bambi","Cinderella",""},
            {"0.00","12.00","15.00","12.00","15.00","18.00","0.00"}, //price
            {"50","50","50","50","50","50","50"} //number of available seats
            };
```

```
introws =0;
rows = movies.length;
System.out.println( rows ) ;                    //number of rows

      System.out.println( movies[0].length ) ;  //number of cols on row 0
      System.out.println( movies[1].length ) ;  //number of cols on row 1
      System.out.println( movies[2].length ) ;  //number of cols on row 2
      System.out.println(movies[3].length) ;    //number of cols on row 3
```

Referring to individual values:

```
System.out.println( movies [0][0] ) ;     // Mon
System.out.println( movies [0][1] ) ;     // Tue
System.out.println( movies [1][3] ) ;     // Rocky
System.out.println( movies [2][3] ) ;     // 12.00
```

Example 2-d array with String values

```
String [ ][ ] movies = {
            {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"},
            {"None","Schrek","Batman","Rocky","Bambi","Rambo","None"},
            {"0.00","12.00","15.00","12.00","15.00","18.00","0.00"}, //price
            {"50","50","50","50","50","50","50"} //number of available seats
            };

      int rows = movies.length;
      int cols = movies[0].length;
      for (int r = 0 ; r <movies.length ;  r++)   {   //loop the row index
            for (int c=0; c<movies[r].length; c ++ ){ //loop the columns of a row
                  System.out.print ( movies[r][c]  +  "\t " ) ;
                  // print the elements on a row
            }
            System.out.println();      // move to the next row
      }
```

Result:

```
Mon   Tue        Wed        Thu        Fri        Sat
None  Schrek     Batman     Rocky      Bambi      Cinderella
0.00  12.00      15.00      12.00      15.00      18.00
```

Example 3: Looping through a two-dimensional array:

```
int rows = movies.length;
int cols = movies[0].length;
for (int col = 0 ; col < cols;  col++)   {
      for (int row=0; row < rows; row ++ ){
            System.out.print ( movies[row][col]  +  "\t " ) ;
      }
      System.out.println();
}
```

Result:

Mon	None	0.00
Tue	Schrek	12.00
Wed	Batman	15.00
Thu	Rocky	12.00
Fri	Bambi	12.00
Sat	Rambo	15.00
Sun	None	18.00

Exercise – MovieTicketsGame V4 – 2d-array

```
String [ ][ ] movies = {
            {"Mon","Tue","Wed","Thu","Fri","Sat","Sun"},
            {"None","Schrek","Batman","Rocky","Bambi","Cinderella",""},
            {"0.00","12.00","15.00","12.00","15.00","18.00","0.00"}, //price
            {"50","50","50","50","50","50","50"} //number of available seats
            };
```

Use the 2-d array above.

1. Ask the end-user to input:
   a. The day of the week,
   b. The number of adults,
   c. The number of children (they pay 50%) and
   d. The number of pensioners (they pay 30% of the adult price).
2. Loop the array , for each occurance:
   a. Test if the day matches the day preferred by the user
   b. Keep the column-index of this day in a variable of type int
3. Display the movie name and price for that column-index
4. Convert the price to a double value
5. Calculate the total ticket price
6. Display the full price to be paid

Exercise : Win a movie ticket

Use the array above.
Generate a random number between 0 and 6
Ask the end-use to guess this number.
They may try 3 guesses
If after 3 guesses they they still have it wrong, say "you lost"
If they got it right, say thet they have won a movie ticket.
The movie thay had won is the one that corresponds to the number that they guessed correctly.
Display this movie name and the day when it shows.

Exercise: Available seats

Loop while there are still seats available
Ask which movie they want to see
Ask how many want to go
Loop to find the column index for that day
If there are enough seats:
Display the corresponding day and price
Convert the price to double, display the price to pay
Subtract the number of people from the number of available seats,
update the array to display number of seats.
Else display – no seats available

2-dimensional with ***Irregular Column*** length:

```
String[][] arrayString = {
        {
        {"actor ","teacher"},
        {"london, ","Bristol","Brighton"},
        {"John ","Peter","Paul  ","Simon","Mike"},
        {"Smith"," Jones "}
        };
```

Get the number of ***ROWS*** in the array:

```
arrayString.length
```

To determine the number of ***COLUMNS*** in a specific row:

```
arrayString[index].length
 arrayString[0].length
arrayString[1].length
arrayString[2].length
arrayString[3].length
```

Looping through a this array:

```
for (int row=0;row<arrayString.length;row++){
        for (int column=0; column <arrayString[row].length; column ++){
                System.out.prints(arrayString[row][column]  + " \t");
        } // end the inner for-loop
        System.out.println();
}// end the outer for-loop
```

//Multi Dimensional Array with irregular column lengths

```
import java.util.Scanner;
public class page17{  //class
public static void main (String[ ] args) {  //main

    int i =0;


    String[][] arrayString =
            {
            {"actor ","teacher"},
            {"london, ","Bristol","Brighton"},
            {"John ","Peter","Paul  ","Simon","Mike"},
            {"Smith"," Jones "}
            };

    // prints the length of the array – how many ROWS are in the array
    System.out.println(arrayString.length + " arrayString.length ");

    // prints the length of each row, i.e. the number of columns per row
    System.out.println("row [0] has " + arrayString[0].length + " " + "columns ");
    System.out.println("row [1] has " + arrayString[1].length + " " + "columns");
    System.out.println("row [2] has " + arrayString[2].length + " " + "columns");
    System.out.println("row [3] has " + arrayString[3].length + " " + "columns");



} //main
} //class
```

Arrays Class

Import java.util.Arrays;

To Sort an array:

```
Arrays.sort(str4);
```

```
// import java.util.Arrays;

String [] str4 = {"x", "c", "v", "b", "n", "s", "d", "f", "g", "h"};
Arrays.sort(str4);
for (String i : str4) {
        System.out.println(i);
}
```

Compare Arrays

Arrays.equals(array1,array2);

```
import java.util.Arrays;
class page19a
{
   public static void main (String[] args)
   {
          int arr1[] = {1, 2, 3};
          int arr2[] = {1, 2, 3};

          if (Arrays.equals(arr1, arr2))
          System.out.println("Same");
          else
           System.out.println("Not same");

  }

}
```

Home work

Look ahead to see how to compare 2 arrays

Your have a String Array with the value {“c”,”i”,”n”,”d”,”e”,”r”, “e”,”l”,”l”,”a”}

And a second array : {“-”,”-”,”-”,”-”,”-”,”-”,”-”, “-”,”-”,”-”,”-”,”-”}

Play the hanman game, filling every correct record into the second array, until the end user guessed 12 times or the 2 arrays are the same.

System.arraycopy(sourcearray,start in source, destination array, start, length)

```
System.arraycopy(x, 1, y, 2, 2);
```

```
System.arraycopy(arr, 0, copied, 1, 5);//5 is the length to copy
```

```
class CopyArray {
    public static void main(String [] args) {
int[] x = {1, 2 ,3};
        int[] y = {4, 5, 6, 7};
        System.arraycopy(x, 1, y, 2, 2);
                        //x is the source,
                        //1 is the starting position in the source,
                        //y is the destination array,
                        //2 is the starting position in the destination array,
                        //2 is the number of array elements to copy
                for(int var : y) {
                    System.out.print(var+" ");
                } //for
    } //main
} //class
```

OUTPUT: 4,5,2,3

System.arraycopy

```
int[] copied =new int[10];
```

Output:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

```
int[] arr ={1,2,3,4,5};
System.arraycopy(arr, 0, copied, 1, 5);//5 is the length to copy
System.out.println(Arrays.toString(copied));
```

Output:
[0, 1, 2, 3, 4, 5, 0, 0, 0, 0]

Arrays.copyOf(arr, lengthNewArr)

```
Arrays.copyOf(arr, 10);
```

```
int[] arr ={1,2,3,4,5};
int[] copied =Arrays.copyOf(arr, 10);//10 the the length of the new array

System.out.println(Arrays.toString(copied));
```

Output:
[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]

```
copied =Arrays.copyOf(arr, 3);

System.out.println(Arrays.toString(copied));
```

Outpout
[1, 2, 3]

Home work

Create the Hangman Game

Home work

Create the win a movie ticket game

Home work

Make the naughts and crosses game

Generics

Generics

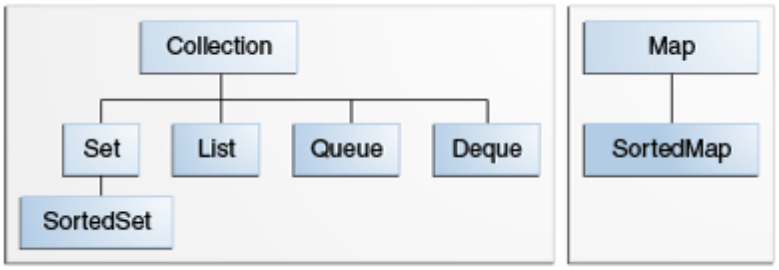
```
public class generics {

public static <E> void printArray (E[] inputArray) {
      // TODO Auto-generated method stub
            for(E element:inputArray){
                  System.out.printf("%s => ",element);
            }
            System.out.println();
}
public static void main (String args[])
{
   Integer[] intArray={1,2,3,4,5};
   Double[] doubleArray={1.1,2.2,3.3,4.4};
   Character[] charArray={'a','b','C'};

   System.out.println ("integer array");
   printArray(intArray);
   System.out.println ("double array");
   printArray(doubleArray);
   System.out.println ("chars array");
   printArray(charArray);

}
}
```

Collections

Collections

Collections Hierarchy

The Collections framework is better understood with the help of **core interfaces**. The collections classes implement these interfaces and provide concrete functionalities.



Java Collections Hierarchy

1.1. Collection

Collection interface is at the root of the hierarchy. Collection interface provides all general purpose methods which all collections classes must support (or throw `UnsupportedOperationException`). It **extends Iterable** interface which adds support for iterating over collection elements using the "**for-each loop**" statement.

All other collection interfaces and classes (except Map) either extend or implement this interface. For example, List ***(indexed, ordered)*** and Set ***(sorted)*** interfaces implement this collection.

1.2. List

Lists represents an **ordered** collection of elements. Using lists, we can access elements by their integer index (position in the list), and search for elements in the list. index start with `0`, just like an array.
Some useful classes which implement `List` interface are – **ArrayList**, **LinkedList**, **Stack** and **Vector**.

1.5. Stack

The Java **Stack** interface represents a classical stack data structure, where elements can be pushed to last-in-first-out (LIFO) stack of objects. In Stack we push an element to the top of the stack, and popped off from the top of the stack again later.

1.3. Set

Sets represents a collection of **sorted** elements. Sets do not allow the duplicate elements. Set interface does not provides no guarantee to return the elements in any predictable order; though some Set implementations store elements in their natural ordering and guarantee this order.

Some useful classes which implement `Set` interface are **HashSet**, **LinkedHashSet** and **TreeSet**.

1.4. Map

The **Map** interface enable us to store data in ***key-value pairs*** (keys should be immutable). A map cannot contain duplicate keys; each key can map to at most one value.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Some useful classes which implement Map interface are **HashMap**, **LinkedHashMap**, **TreeMap**

1.6. Queue

A queue data structure is intended to hold the elements (put by producer threads) prior to processing by consumer thread(s). Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. One such exception is priority queue which order elements according to a supplied Comparator, or the elements' natural ordering.

In general, queues do not support blocking insertion or retrieval operations. Blocking queue implementations classes implement **BlockingQueue** interface.

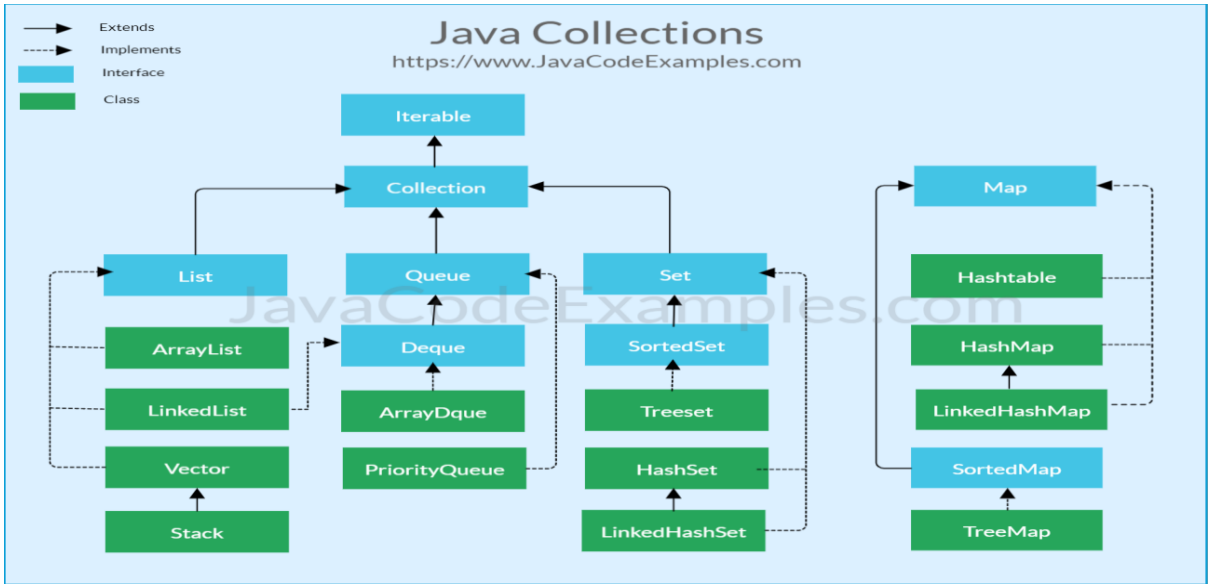
Some useful classes which implement Map interface are – ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue and SynchronousQueue.

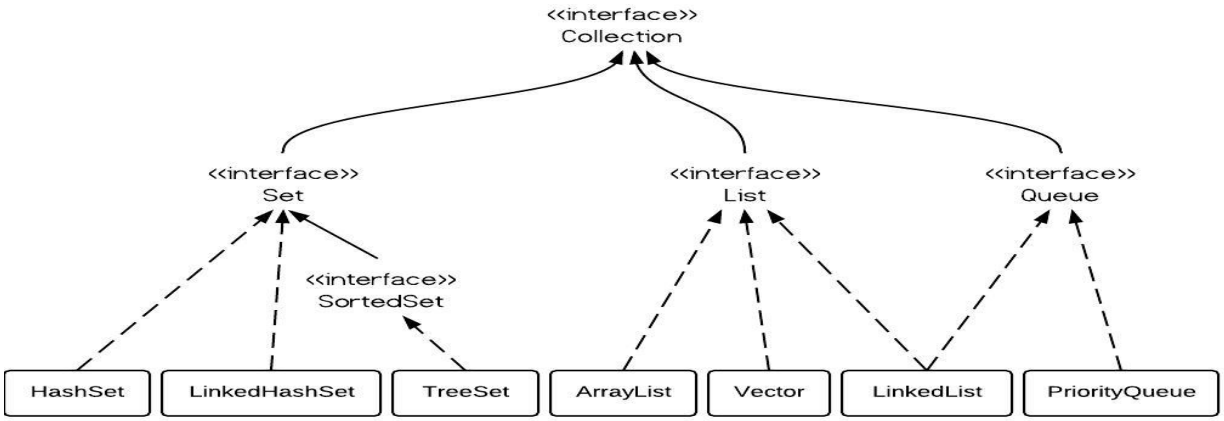
1.7. Deque

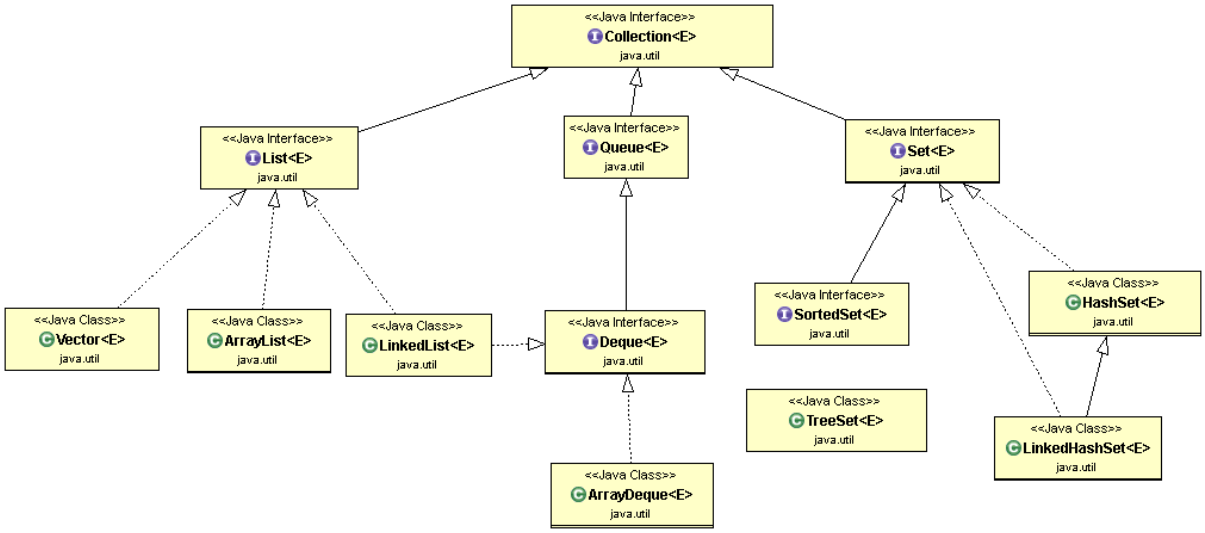
A double ended queue (pronounced "*deck*") that supports element insertion and removal at both ends. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. When a deque is used as a stack, LIFO (Last-In-First-Out) behavior results.

This interface should be used in preference to the legacy Stack class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.

Some common known classes implementing this interface are ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque and LinkedList.







Interface	Has Duplicates?	Performance	Implementations			Historical
List	yes	Fast on … Slow on …	**ArrayList** ***Insertion order***	LinkedList Insertion Order		Vector, Stack
Set	no	Fast on … Slow on …	HashSet *Undefined order*	**LinkedHashSet** ***insertion order***	TreeSet *Ascending Ordered*	…
Queue	yes	Fast on … Slow on …	BlockingQueue with ArrayBlockingQueue	Queue with LinkedList	PriorityQueue	
Deque	yes	Fast on … Slow on …	Deque with ArrayDeque			
Map	no duplicate keys	Fast on … Slow on …	HashMap ***Undefined order***	**LinkedHashMap** ***Insertion order of keys***	TreeMap ***Ascending order of keys***	Hashtable

Basic approach to choosing a collection

The overall approach I'd suggest for choosing is as follows:

1. choose the **general type** of organisation / order that your data needs to have
2. Do you need duplicates
3. then, choose the implementation of that type that has the **minimum functionality** that you actually require (e.g. don't choose a sorted structure if you don't actually need the data to be sorted).
4. How large is your dataset and how fast is your collection, i.e. will is slow down your application

In general, the algorithm that underlies each collection class is designed to be a good tradeoff between efficiency and certain minimal requirements. So as long as you understand the *minimal requirements* that a given class is designed to provide, you shouldn't need to get too bogged down in the actual algorithms (though if you *are* interested in algorithms, the source code to all the collections classes is available and they make fascinating case studies, of course).

ArrayList. Overview

ArrayList Features

ArrayList has the following features –

1. **Ordered** – Elements in arraylist preserve their ordering which is by default the order in which they were added to the list.
2. **Index based** – Elements can be randomly accessed using index positions. Index start with '0'.
3. **Dynamic resizing** – ArrayList grows dynamically when more elements needs to be added than it's current size.
4. **Duplicates allowed** – We can add duplicate elements in arraylist. It is not possible in sets.

ArrayList Example

Declare:

```
List<String> listWords = new ArrayList<String>();
ArrayList <String> listWords = new ArrayList<String>();
List<Integer> listNumbers = new ArrayList<Integer>();
List<String> linkedWords = new LinkedList<String>();
```

Size it:

```
List<Integer> listNumbers = new ArrayList<>(1000);  // up to 1000 elements
```

Add:

`add(Object)`, **`add(index, Object)`** and **`addAll()`**

Retrieve:

```
Number number = linkedNumbers.get(3);
Number first = numbers.getFirst();
Number last = numbers.getLast();
```

- **`boolean contains(Object)`**: returns `true` if the list contains the specified element.
- **`int indexOf(Object)`**: returns the index of the first occurrence of the specified element in the list, or -1 if the element is not found.
- **`int lastIndexOf(Object)`**:

Update:

`set(index, element)`

Remove:

`remove(index)` or **`remove(Object)`**

```
listStrings.clear();
```

Adding values to the ArrayList - Using integers

```
ArrayList<Integer> myArrayList = new ArrayList<Integer>();
myArrayList.add(3);
myArrayList.add(2);
myArrayList.add(1);
myArrayList.add(4);
myArrayList.add(5);
myArrayList.add(6);
myArrayList.add(6);

Iterator<Integer> myIterator = myArrayList.iterator();
while(myIterator.hasNext()){
        System.out.println(myIterator.next());
}
```

Adding values to the ArrayList - Using String

```
ArrayList<String> aListString = new ArrayList<String>();
        aListString.add("adam");
        aListString.add("peter");
        aListString.add("paul");
```

Loop through the arraylist using Iterator

```
Iterator<String> myIterator = aListString.iterator();
        while(myIterator.hasNext()){
                System.out.print(myIterator.next() + " ");
        }
        System.out.println();
        System.out.print("get2 " + aListString.get(2));
```

/*Add element to an array list at the given index*/

```
myArrayList.add(0, "Rahul");
myArrayList.add(1, "Justin");
System.out.println("Currently the arrayList is a :"+myArrayList );
```

/*Remove elements from array List*/

```
myArrayList.remove(1);   // at position 1
myArrayList.remove("Rahul");   //remove this value
String name = “Justin”;
myArrayList.remove(name);   //remove this value
    System.out.println("Currently the arrayList is c :"+myArrayList );
```

/* show the size of the array list*/

```
    System.out.println("number of elements in the arrayList is : "+ myArrayList.size());
```

Looping in ArrayLists

```
ArrayList<Integer> myArrayList = new ArrayList<Integer>();
myArrayList.add(14);
myArrayList.add(7);
myArrayList.add(39);
myArrayList.add(40);
```

/* For Loop for iterating ArrayList */

```
System.out.println("For Loop");
for (int counter = 0; counter < myArrayList.size(); counter++) {
   System.out.println(myArrayList.get(counter));
}
```

/* Advanced For Loop*/

```
System.out.println("Advanced For Loop");
for (Integer num : myArrayList) {
    System.out.println(num);
}
```

/* While Loop for iterating ArrayList*/

```
System.out.println("While Loop");
int count = 0;
while (myArrayList.size() > count) {
  System.out.println(myArrayList.get(count));
   count++;
}
```

/*Looping Array List using Iterator*/

```
System.out.println("Iterator");
Iterator myIterator= myArrayList.iterator();
while (myIterator.hasNext()) {
            System.out.println(myIterator.next());
 }
```

Joing or Compbine Arraylists

```
//First ArrayList
ArrayList<String> arraylist1=new ArrayList<String>();
arraylist1.add("AL1: tom");
arraylist1.add("AL1: dick");
arraylist1.add("AL1: harry");

//Second ArrayList
ArrayList<String> arraylist2=new ArrayList<String>();
arraylist2.add("AL2: allie");
arraylist2.add("AL2: bella");
arraylist2.add("AL2: zoey");

//New ArrayList
ArrayList<String> myArrayList = new ArrayList<String>();
myArrayList.addAll(arraylist1);
myArrayList.addAll(arraylist2);


//Displaying elements of the joined ArrayList
for(String temp: myArrayList){
   System.out.println(temp);
}
```

Comparing Arraylists

```
ArrayList<String> myArrayList1= new ArrayList<String>();
    myArrayList1.add("hi");
    myArrayList1.add("How are you");
    myArrayList1.add("Good Morning");
    myArrayList1.add("bye");
    myArrayList1.add("Good night");

    ArrayList<String> myArrayList2= new ArrayList<String>();
    myArrayList2.add("Howdy");
    myArrayList2.add("Good Evening");
    myArrayList2.add("bye");
    myArrayList2.add("Good night");

    //Storing the comparison output in ArrayList<String>
    ArrayList<String> myArrayList3= new ArrayList<String>();
    for (String temp : myArrayList1)
       myArrayList3.add(myArrayList2.contains(temp) ? "Yes" : "No");
    System.out.println(myArrayList3);

    //Storing the comparison output in ArrayList<Integer>
    ArrayList<Integer> myArrayList4= new ArrayList<Integer>();
    for (String temp2 : myArrayList1)
        myArrayList4.add(myArrayList2.contains(temp2) ? 1 : 0);
          System.out.println myArrayList 4);
```

ArrayList exercise

Amadeus,Drama,160 Mins.,1984,14.83
As Good As It Gets,Drama,139 Mins.,1998,11.3
Batman,Action,126 Mins.,1989,10.15
Billy Elliot,Drama,111 Mins.,2001,10.23
Blade Runner, Science Fiction,117 Mins.,1982,11.98
Shadowlands,Drama,133 Mins.,1993,9.89
Shrek,Animation,93 Mins,2001,15.99
Snatch,Action,103 Mins,2001,20.67
The Lord of the Rings,Fantasy,178 Mins,2001,25.87

Read the above file into an array

Create a class: fileUtilities
Method getFileSize

- input paratmeter – String filename
- returns an integer recordCount
- to do :
  - read a file and
  - count the records

Method createArray

- input parameters:
  - an empty array String [] myArray (size has been allocated to the array) ,
  - String Filename
- returns the populated myArray
- To Do :
  - read a file
  - for every record, transfer the record into the array

Method createArrayList

- input parameters:
  - String Filename
- returns the populated myArrayList
- To Do :
  - read a file
  - for every record, transfer the record into the arrayList , ArrayList<String> myArrayList

Method displayArray

- input parameter – String[] myArray
- returns void
- To Do : loop through the end of the myArray and display all the elements

Method displayArrayList
- input parameter – ArrayList<String> myArrayList
- returns void
- To Do : loop through the end of the myArrayList and display all the elements

Method findMovieArray
- input parameters – the array and the name of a movie
- returns void
- to do :
  - loop through the end of the Array
  - if the movie is found, display "found ", else display "not found"

Method findMovieArrayList
- input parameters – the arrayList and the name of a movie
- returns void
- to do :
  - loop through the end of the ArrayList
  - if the movie is found, display "found ", else display "not found"

Method insertMovieArrayList
- input parameter – the arrayList and the name of a movie
- returns void
- loop through the end of the ArrayList
  - if the movie is found, display "found "
  - else display "not found", and add this movie to the arrayList

Method deleteMovieArrayList
- input parameter – the arrayList and the name of a movie
- returns void
- loop through the end of the ArrayList
  - if the movie is found, display "found ", and remove this movie from the arrayList
  - else display "not found"

Executing class with Main method

- get the size of a file, create and array with this size, Read the file into an array of this size
- Display the array
- Read the file into an arrayList
- Display the arrayList
- Get the name of a movie from the end-user
- Find this movie on the array
- Find this movie on the Arraylist
- If not found, add it into the arraylist and display the arraylist
- If found, remove it from the array list and display the array list

Initialise an ArrayList

```
ArrayList<String> myArrayList = new ArrayList<String>(
                    Arrays.asList("Paul", "Peter", "Mike"));
          System.out.println("Elements are:"+ myArrayList);
```

Exercise initialize and arry List

Create the above arrayLst
Create an output file of filetype txt, with a record for each element on the arraylist

Sort an ArrayList

```
ArrayList<String> myArralyList = new ArrayList<String>();
myArralyList.add("India");
myArralyList.add("US");
myArralyList.add("China");
myArralyList.add("Denmark");

/*Unsorted List*/
System.out.println("Before Sorting:");
for(String temp : myArralyList){
        System.out.println(temp);
    }

/* Sort statement*/
Collections.sort(myArralyList);

/* Sorted List*/
System.out.println("After Sorting:");
for(String temp: myArralyList){
        System.out.println(temp);
    }
```

Exercise sort an arraylis

Creata a program:
Initialise a String moreFriends = “no”;
Loop till the end-user says “no”

- Ask the end –user what the name of his friend is
- Read the name, and add it to an array list
- Ask the end-user if he has another friend

When all friends’ names are read, sort the array and display the names in alphabetical order

Array to Array List

Method 1

```
/* Array Declaration and initialization*/
        String citynames[]={"Agra", "Mysore", "Chandigarh", "Bhopal"};

        /*Array to ArrayList conversion*/
        ArrayList<String> citylist= new ArrayList<String>(Arrays.asList(citynames));

        /*Adding new elements to the converted List*/
        citylist.add("New City2");
        citylist.add("New City3");

        /*Final ArrayList content display using for*/
        for (String str: citylist)
        {
                System.out.println(str);
                        }
```

Method 2

```
/* Array Declaration and initialization*/
        String array[]={"Hi", "Hello", "Howdy", "Bye"};

        /*ArrayList declaration*/
        ArrayList<String> arraylist= new ArrayList<String>();

        /*Conversion*/
        Collections.addAll(arraylist, array);

        /*Adding new elements to the converted List*/
        arraylist.add("String1");
        arraylist.add("String2");

        /*Display array list*/
        for (String str: arraylist)
        {
                System.out.println(str);
        }
```

ArrayList to Array

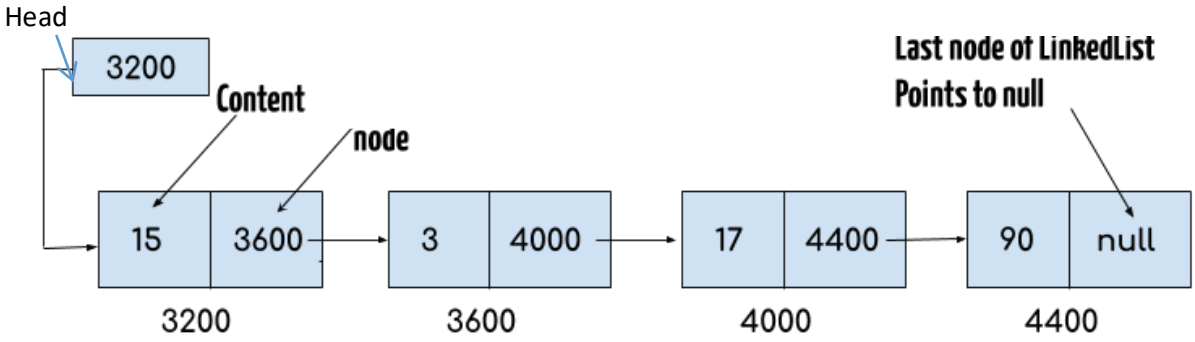
Method 1 get()

```
/*ArrayList declaration and initialization*/
ArrayList<String> arrlist= new ArrayList<String>();
arrlist.add("String1");
arrlist.add("String2");
arrlist.add("String3");
arrlist.add("String4");


/*ArrayList to Array Conversion */
String array[] = new String[arrlist.size()];
for(int j =0;j<arrlist.size();j++){
  array[j] = arrlist.get(j);
}


/*Displaying Array elements*/
for(String k: array)
{
        System.out.println(k);
}
```

Method 2 - toArray

```
/*ArrayList declaration and initialization*/
ArrayList<String> friendsnames= new ArrayList<String>();
friendsnames.add("Ankur");
friendsnames.add("Ajeet");
friendsnames.add("Harsh");
friendsnames.add("John");

/*ArrayList to Array Conversion */
String frnames[]=friendsnames.toArray(new String[friendsnames.size()]);

/*Displaying Array elements*/
for(String k: frnames)
{
        System.out.println(k);
}
```

LinkedList Overview



LinkedList Features

- Permits all elements including duplicates and NULL.
- LinkedList maintains the **insertion order** of the elements.
- It does not implement RandomAccess interface. So we can access elements in sequential order only. It does not support accessing elements randomly.
- We can use ListIterator to iterate LinkedList elements.

    - A doubly linked list (often abbreviated as DLL) is very much like a regular singly linked list (SLL).
      Both DLL and SLL contain a pointer to the next node, as well as a data field to represent the actual value stored in the node.
    - However, the difference between DLL and SLL is that the doubly linked list also contains a pointer to the previous node, not just the next node.

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

LinkedList Methods

boolean add(Object o) : appends the specified element to the end of a list.
void add(int index, Object element) : inserts the specified element at the specified position index in a list.
void addFirst(Object o) : inserts the given element at the beginning of a list.
void addLast(Object o) : appends the given element to the end of a list.
int size() : returns the number of elements in a list
boolean contains(Object o) : return true if the list contains a specified element, else false.
boolean remove(Object o) : removes the first occurence of the specified element in a list.
Object getFirst() : returns the first element in a list.
Object getLast() : returns the last element in a list.
int indexOf(Object o) : returns the index in a list of the first occurrence of the specified element, or -1 if the list does not contain specified element.
lastIndexOf(Object o) : returns the index in a list of the last occurrence of the specified element, or -1 if the list does not contain specified element.
Iterator iterator() : returns an iterator over the elements in this list in proper sequence.

LinkedList Example

```
LinkedList<Integer> ll = new LinkedList<Integer>();

ll.add(3);
ll.add(2);
ll.add(1);
ll.add(4);
ll.add(5);
ll.add(6);
ll.add(6);

Iterator<Integer> iter2 = ll.iterator();
    while(iter2.hasNext()){
    System.out.println(iter2.next());
}

System.out.println("ll.get(2) " + ll.get(2));

System.out.println("Linked List: traversing elements in backward
direction...");
            while(itr2.hasPrevious()){
            System.out.println(itr2.previous());
      }
```

Vectors Methods

Vectors Methods

Remove, contains, size,

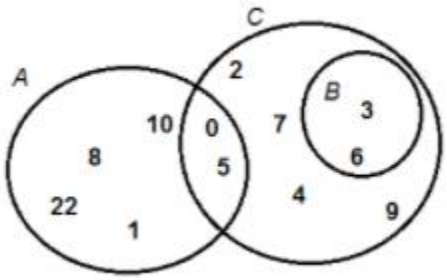
Vectors Example

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

```
 Vector<String> v=new Vector<String>();//creating vector
         //add
         v.add("umesh");//method of Collection
         v.addElement("irfan");//method of Vector
         v.addElement("kumar");
        //get
         System.out.println("v.get(2) " + v.get(2));
         System.out.println("v.contains umesh " +
v.contains("umesh"));
         v.remove("umesh");
         System.out.println("v.size() " + v.size());
         System.out.println("v.isempty() " + v.isEmpty());
         System.out.println("v.firstElement() " + v.firstElement());
         System.out.println("v.lastElement() " + v.lastElement());
         v.clear();
         System.out.println("v.isempty() " + v.isEmpty());

         //traversing elements using Enumeration
         Enumeration e=v.elements();
                  while(e.hasMoreElements()){
            System.out.println(e.nextElement());
         }
```

Overview of Set Collection

Basically, **Set** is a type of collection that does not allow duplicate elements. That means an element can only exist once in a Set. It models the set abstraction in mathematics. The following picture illustrates three sets of numbers in mathematics:



Characteristics of a Set collection:

The following characteristics differentiate a Set collection from others in the Java Collections framework:

- Duplicate elements are not allowed.
- Elements are not stored in order. That means you cannot expect elements sorted in any order when iterating over elements of a Set.

Why and When Use Sets?

Based on the characteristics, consider using a Set collection when:

- You want to store elements distinctly without duplication, or unique elements.
- You don't care about the order of elements.

For example, you can use a Set to store unique integer numbers; you can use a Set to store cards randomly in a card game; you can use a Set to store numbers in random order, etc.

HashSet vs. TreeSet vs. LinkedHashSet

There are 3 commonly used implementations of Set: HashSet, TreeSet and LinkedHashSet.

A Set contains no duplicate elements. You can simply add elements to a set, and duplicates will be removed automatically.

Compare in brief:

- Set interface extends Collection interface.
- if you need a fast set, you should use HashSet;
- if you need a sorted set, then TreeSet should be used;
- if you need a set that can be store the insertion order, LinkedHashSet should be used.

The elements in a set are sorted, but the add, remove, and contains methods has time complexity of O(log (n)).

It offers several methods to deal with the ordered set like :

- first(), last(), headSet(), tailSet(), etc.

LinkedHashSet Overview

LinkedHashSet Features

- It extends `HashSet` class which extends `AbstractSet` class.
- It implements `Set` interface.
- **Duplicate values are not allowed** in LinkedHashSet.
- One NULL element is allowed in LinkedHashSet.
- It is an **ordered collection** which is the order in which elements were inserted into the set (**insertion-order**).
- basic operations(add, remove, contains and size).

LinkedHashSet Methods

add ,boolean contains(Object o)
boolean remove(Object o), void clear(),(no get , iterate , to get at position convert to array)
first(), last(), headSet(), tailSet(), etc.
int size(), boolean isEmpty(),
Iterate

LinkedHashSet Example

```
//contains unique elements only like HashSet.
//It extends HashSet class and implements Set interface.
//maintains insertion order.

System.out.println();
System.out.println("LinkedHashsets");
	LinkedHashSet<String> alLHS=new LinkedHashSet<String>();
	alLHS.add("Ravi");
	alLHS.add("Vijay");
	alLHS.add("Ravi");
	alLHS.add("Ajay");
	Iterator<String> itrLHS=alLHS.iterator();
	while(itrLHS.hasNext()){
		System.out.println(itrLHS.next());
	}
```

Hashset Overview

HashSet Features

- It implements `Set` Interface.
- Duplicate values are not allowed in HashSet.
- One NULL element is allowed in HashSet.
- It is un-ordered collection and makes no guarantees as to the iteration order of the set.

HashSet Methods

Add, contains , size, remove, isempty (no get , iterate , to get at position convert to eg array)

Hashset Example

```
System.out.println();
System.out.println("Hashsets");

      HashSet<String> alHS=new HashSet<String>();
      alHS.add("Samson");
      alHS.add("Benjamin");
      alHS.add("Jonathan");
      alHS.add("Samson");

      Iterator<String> itrHS=alHS.iterator();
      while(itrHS.hasNext()){
                  System.out.println(itrHS.next());
      }

      //System.out.println(alHS.get(2));
      System.out.println(alHS); // print all
      System.out.println( "Empty? " + alHS.isEmpty());
      System.out.println( "Contains? " + alHS.contains("Benjamin"));
      System.out.println( "Size? "+ alHS.size());
      alHS.remove("Benjamin");
      System.out.println( "Contains? " + alHS.contains("Benjamin"));
      System.out.println( "Size? "+ alHS.size());
      alHS.clear();
      System.out.println( "Empty? " + alHS.isEmpty());
```

TreeSet Overview

TreeSet Features

- It extends AbstractSet class which extends AbstractCollection class.
- It implements NavigableSet interface which extends SortedSet interface.
- Duplicate values are not allowed in TreeSet.
- NULL is not allowed in TreeSet.
- It is an **ordered collection** which store the elements in sorted order.
- Like HashSet, this class offers constant time performance for the basic operations(add, remove, contains and size).
- TreeSet does not allow to insert heterogeneous objects because it must compare objects to determine sort order.

TreeSet Methods

Add, First , last, clear, contains, isempty size, remove

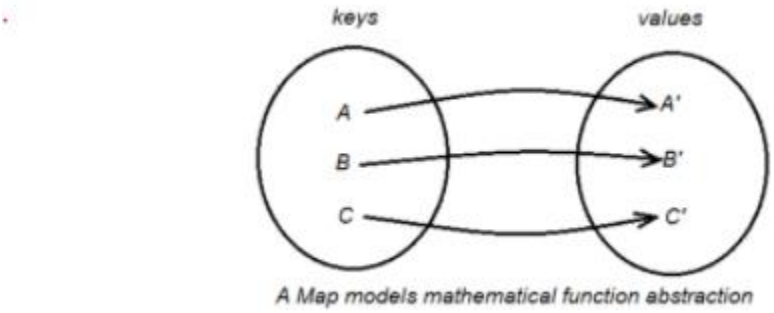
TreeSet Example

```
import java.util.*;
public class buckyTreeSet {

   public static void main(String args[]) {
      // Create a tree set
      TreeSet ts = new TreeSet();
      // Add elements to the tree set
      ts.add("C");
      ts.add("A");
      ts.add("B");
      ts.add("E");
      ts.add("F");
      ts.add("D");
      System.out.println(ts);

   Iterator<String> iterString = ts.iterator();
                  while(iterString.hasNext()){
                        System.out.print(iterString.next() + " ");
                  }

   }
}
```

Overview of Map Collection

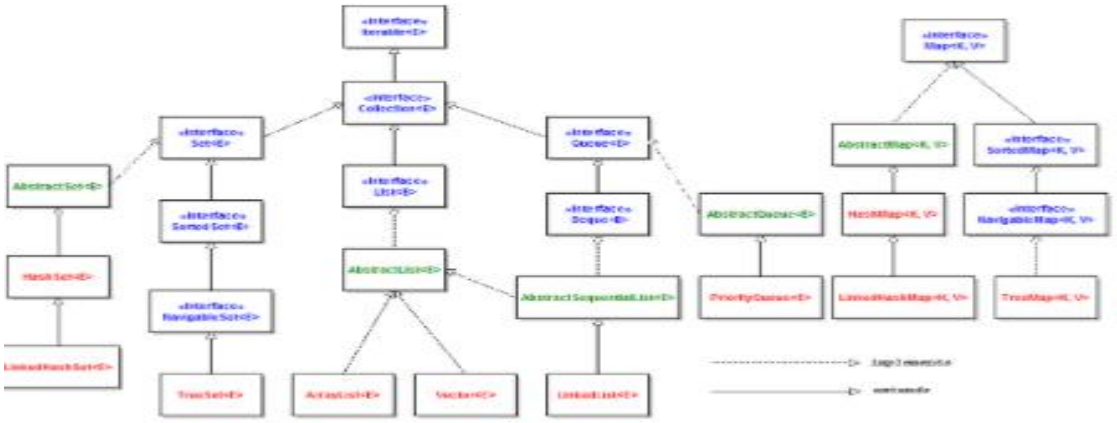
1. If you need `SortedMap` operations or key-ordered `Collection`-view iteration, use `TreeMap`;
2. if you want maximum speed and don't care about iteration order, use `HashMap`
3. if you want near-`HashMap` performance and insertion-order iteration, use `LinkedHashMap`.

A **Map** is an object that maps keys to values, or is a collection of attribute-value pairs. It models the function abstraction in mathematics. The following picture illustrates a map:



A Map models mathematical function abstraction

Note that a Map is not considered to be a true collection, as the Map interface does not extend the Collection interface. Instead, it starts an independent branch in the Java Collections Framework, as shown in the following diagram:



Characteristics of a Map:

Because a Map is not a true collection, its characteristics and behaviors are different than the other collections like List or Set.
A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value (HashMap and LinkedHashMap) but some does not (TreeMap).
The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.

Why and When Use Maps:

Maps are perfectly for key-value association mapping such as dictionaries. Use Maps when you want to retrieve and update elements by keys, or perform lookups by keys. Some examples:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

This tutorial provides code examples around the three major implementations of Map which are described below.

.

HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
7) Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
8) HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

HashMap Overview

Java HashMap Features

- HashMap cannot contain duplicate keys.
- HashMap allows multiple `null` values but only one `null` key.
- HashMap is an **unordered collection**. It does not guarantee any specific order of the elements.
- A value can be retrieved only using the associated key.
- HashMap stores only object references. So primitives must be used with their corresponding wrapper classes. Such as `int` will be stored as `Integer`.

Hashmap Methods

Put, get, size, iterrate

HashMap: Example

Always use interface type (`Map`), generics and diamond operator to declare a new map. The following code creates a `HashMap`

```
Map<Integer, String> mapHttp = new HashMap<>();

       mapHttp.put(200, "OK");
       mapHttp.put(303, "See Other");
       mapHttp.put(404, "Not Found");
       mapHttp.put(500, "Internal Server Error");

       System.out.println(mapHttp);

       System.out.println("Size of IdentintyHashMap: "+mapHttp.size());
       String b = mapHttp.get("Amy");


           System.out.println("Displaying values from HashMap...");
             for (Integer i: mapHttp.keySet()) {
                System.out.println(mapHttp.get(i));
             }
```

LinkedHashMap Overview

LinkedHashMap Features

The important things to learn about Java LinkedHashMap class are:

- It stores key-value pairs similar to HashMap.
- It contains only unique keys. Duplicate keys are not allowed.
- It may have one `null` key and multiple `null` values.
- It maintains the order of K,V pairs inserted to it by adding elements to internally managed **doubly-linked list**.

Methods

void clear() Removes all mappings from this map.

boolean containsKey(Object key) Returns true if this map maps one or more keys to the specified value.

Object get(Object key) Returns the value to which this map maps the specified key.

protected boolean removeEldestEntry(Map.Entry eldest) Returns true if this map should remove its eldest entry.

LinkedHashMap: Example

The following code creates a `LinkedHashMap` that maps phone numbers with contact names:

```
Map<String, String> mapContacts = new LinkedHashMap<>();

mapContacts.put("0169238175", "Tom");
mapContacts.put("0904891321", "Peter");
mapContacts.put("0945678912", "Mary");
mapContacts.put("0981127421", "John");

System.out.println(mapContacts);
```

TreeMap Overview

. TreeMap Features

- It stores key-value pairs similar to like HashMap.
- It allows only distinct keys. Duplicate keys are not possible.
- It cannot have null key but can have multiple null values.
- It stores the keys in sorted order (natural order) It provides guaranteed **log(n)** time cost for the containsKey, get, put and remove operations.

Methods

void clear() Removes all mappings from this TreeMap.
Object clone() Returns a shallow copy of this TreeMap instance.
Comparator comparator() Returns the comparator used to order this map, or null if this map uses its keys' natural order.
boolean containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set entrySet() Returns a set view of the mappings contained in this map.
Object firstKey() Returns the first (lowest) key currently in this sorted map.
Object get(Object key) Returns the value to which this map maps the specified key.
SortedMap headMap(Object toKey) Returns a view of the portion of this map whose keys are strictly less than toKey.
Set keySet() Returns a Set view of the keys contained in this map.
Object lastKey() Returns the last (highest) key currently in this sorted map.
Object put(Object key, Object value) Associates the specified value with the specified key in this map.
void putAll(Map map) Copies all of the mappings from the specified map to this map.
Object remove(Object key) Removes the mapping for this key from this TreeMap if present.
int size() Returns the number of key-value mappings in this map.
SortedMap subMap(Object fromKey, Object toKey) Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap tailMap(Object fromKey) Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
Collection values() Returns a collection view of the values contained in this map.

TreeMap Example:

The following code creates a TreeMap that maps file extensions to programming languages:

```
Map<String, String> mapLang = new TreeMap<>();

mapLang.put(".c", "C");
mapLang.put(".java", "Java");
mapLang.put(".pl", "Perl");
mapLang.put(".cs", "C#");
mapLang.put(".php", "PHP");
mapLang.put(".cpp", "C++");
mapLang.put(".xml", "XML");

System.out.println(mapLang);
```

Queue Overview

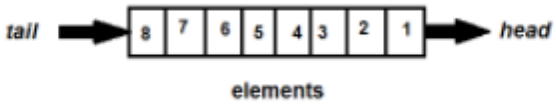
1. What is Queue?

Queue means 'waiting line', which is very similar to queues in real life: a queue of people standing in an airport's check-in gate; a queue of cars waiting for green light in a road in the city; a queue of customers waiting to be served in a bank's counter, etc.

In programming, queue is a data structure that holds elements prior to processing, similar to queues in real-life scenarios. Let's consider a queue holds a list of waiting customers in a bank's counter. Each customer is served one after another, follow the order they appear or registered. The first customer comes is served first, and after him is the 2nd, the 3rd, and so on. When serving a customer is done, he or she lefts the counter (removed from the queue), and the next customer is picked to be served next. Other customers come later are added to the end of the queue. This processing is called First In First Out or FIFO.

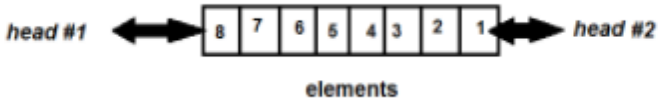
2. Characteristics of Queue

Basically, a queue has a head and a tail. New elements are added to the tail, and to-be-processed elements are picked from the head. The following picture illustrates a typical queue:



Elements in the queue are maintained by their insertion order. The Queue interface abstracts this kind of queue.

Another kind of queue is double ended queue, or deque. A deque has two heads, allowing elements to be added or removed from both ends. The following picture illustrates this kind of queue:



The Deque interface abstracts this kind of queue, and it is a sub interface of the Queue interface. And the LinkedList class is a well-known implementation.

Queue does allow duplicate elements, because the primary characteristic of queue is maintaining elements by their insertion order. Duplicate elements in terms of equals contract are considered distinct in terms of queue, as there is no two elements having same ordering.

3. Behaviors of Queue

Due to Queue's nature, the key operations that differentiate Queue from other collections are extraction and inspection at the head of the queue.

For deques, the extraction and inspection can be processed on both ends.

And because the Queue interface extends the Collection interface, all Queue implementations provide core operations of a collection like add(), contains(), remove(), clear(), isEmpty(), etc.

And keep in mind that, with queues, operations on the head are fastest (e.g. offer() and remove()), whereas operations on middle elements are slow (e.g. contains(e) and remove(e)).

4. Queue's Interfaces

Queue is the super interface of the queue branch in the Java Collection Framework. Under it, there are the following sub interfaces:

- **Deque**: abstracts a queue that has two heads. A deque allows adding or removing elements at both ends.
- **BlockingQueue**: abstracts a type of queues that waits for the queue to be non-empty when retrieving an element, and waits for space to become available in the queue when storing an element.
- **BlockingDeque**: is similar to BlockingQueue, but for double ended queues. It is sub interface of the BlockingQueue

5. Major Queue's Implementations

+ **LinkedList**: this class implements both List and Deque interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a LinkedList when you want fast adding and fast removing elements at both ends, plus accessing elements by index.

+ **PriorityQueue**: this queue orders elements according to their natural ordering, or by a Comparator provided at construction time. Consider using a PriorityQueue when you want to take advantages of natural ordering and fast adding elements to the tail and fast removing elements at the head of the queue.

+ **ArrayDeque**: a simple implementation of the Deque interface. Consider using an ArrayDeque when you want to utilize features of a double ended queue without list-based ones (simpler than a LinkedList).

Queue with linkedlist

Queue with linkedlist methods

Methods: add, contains, soze, clear, isEmpty, Ietrrator

Queue with linkedlist example

```
Queue<String> namesQueue = new LinkedList<>();

System.out.println(queueNames);

	namesQueue.add("Blue");
	namesQueue.add("Green");
	namesQueue.contains("Green");
	namesQueue.remove("Blue");
//		namesQueue.set("Green");
//		namesQueue.get("Blue");
	namesQueue.size();
	namesQueue.clear();
	namesQueue.isEmpty();
	Iterator<String> myIterator = namesQueue.iterator();
	while(myIterator.hasNext()){
		System.out.println(myIterator.next());
	}
```

FIFO(First In First Out) order.

Queue with Linked List

Queue with Linked List methods

Methods: Add, contains, remove, isempty, size, clear
Iterrator

Queue with Linked List example

```
List<String> listNames = Arrays.asList("Alice", "Bob", "Cole",
"Dale", "Eric", "Frank");

Queue<String> queueNames = new LinkedList<>(listNames);

	queueNames.add("Blue");
	queueNames.contains("Blue");
	queueNames.remove("Blue");
	queueNames.isEmpty();
	queueNames.size();
	queueNames.clear();
	queueNames.isEmpty();
	Iterator<String> myIterator4 = queueNames.iterator();
	while(myIterator4.hasNext()){
		System.out.println(myIterator.next());
	}
```

ArrayBlockingQueue Overview

ArrayBlockingQueue class is Java **concurrent** and **bounded** blocking queue implementation backed by an array. It orders elements FIFO (first-in-first-out).

The **head** of the ArrayBlockingQueue is that element that has been on the queue the longest time. The **tail** of the ArrayBlockingQueue is that element that has been on the queue the shortest time. New **elements are inserted at the tail** of the queue, and the queue **retrieval operations obtain elements at the head** of the queue.

ArrayBlockingQueue Features

Let's note down few important points on the ArrayBlockingQueue class.

- ArrayBlockingQueue is a bounded queue of fixed size backed by an array.
- It orders elements FIFO (first-in-first-out).
- Elements are inserted at the tail, and retrieved from the head of the queue.
- Once created, the capacity of the queue cannot be changed.
- It supplies **blocking insertion and retrieval operations**.
- It does not allow NULL objects.
- The Iterator provided in method **iterator()** traverse the elements in order from first (head) to last (tail).

ArrayBlockingQueue Methods

Methods: Add, contains, remove, isempty, size, clear
Iterrator

ArrayBlockingQueue Example

```
BlockingQueue<String> waitingCustomers = new
ArrayBlockingQueue<>(100);
	waitingCustomers.add("Blue");
	waitingCustomers.add("Blue");
	waitingCustomers.remove("Blue");
	waitingCustomers.contains(2);
	waitingCustomers.contains("Blue");
//		numbersDeque.set(0,1);
//		numbersDeque.get(0);
	waitingCustomers.size();
	waitingCustomers.clear();
	waitingCustomers.isEmpty();
	Iterator<String> myIterator3 = waitingCustomers.iterator();
	while(myIterator3.hasNext()){
		System.out.println(myIterator.next());
	}
```

ArrayDeque Overview

The **ArrayDeque** in Java provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as ***Array Double Ended Queue*** or ***Array Deck***. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.

ArrayDeque Features

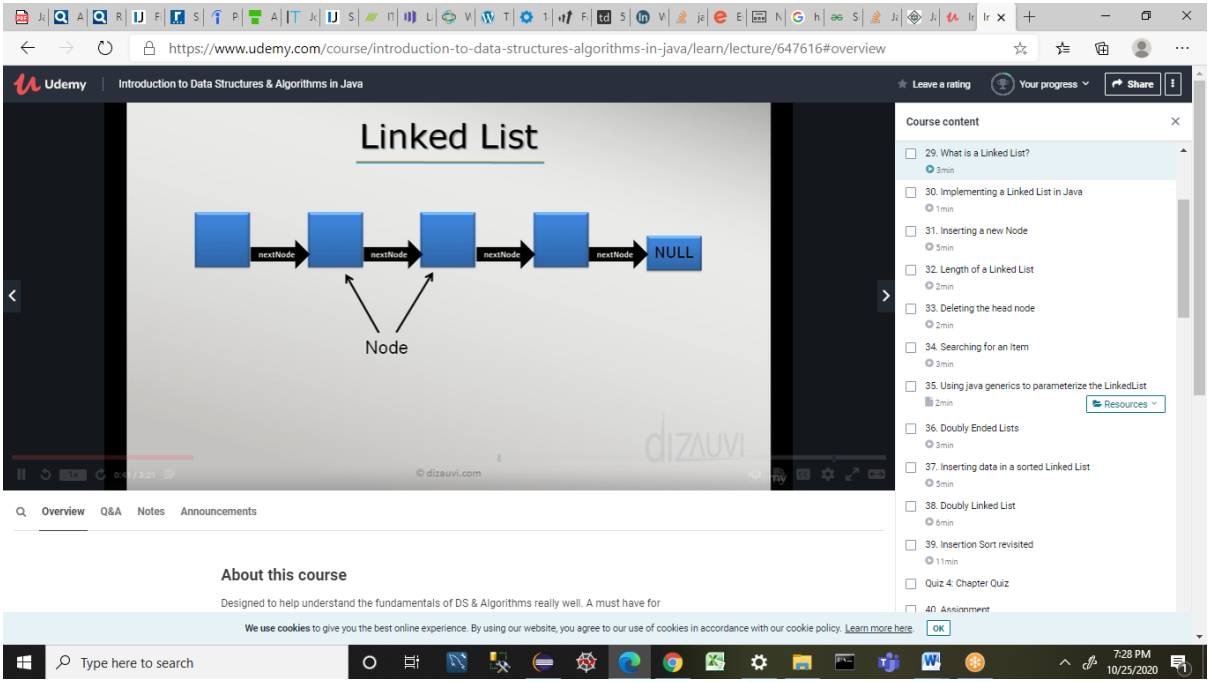
- Array deques have no capacity restrictions and they grow as necessary to support usage.
- They are not thread-safe which means that in the absence of external synchronization, ArrayDeque does not support concurrent access by multiple threads.
- Null elements are prohibited in the ArrayDeque.
- ArrayDeque class is likely to be faster than Stack when used as a stack.
- ArrayDeque class is likely to be faster than LinkedList when used as a queue.

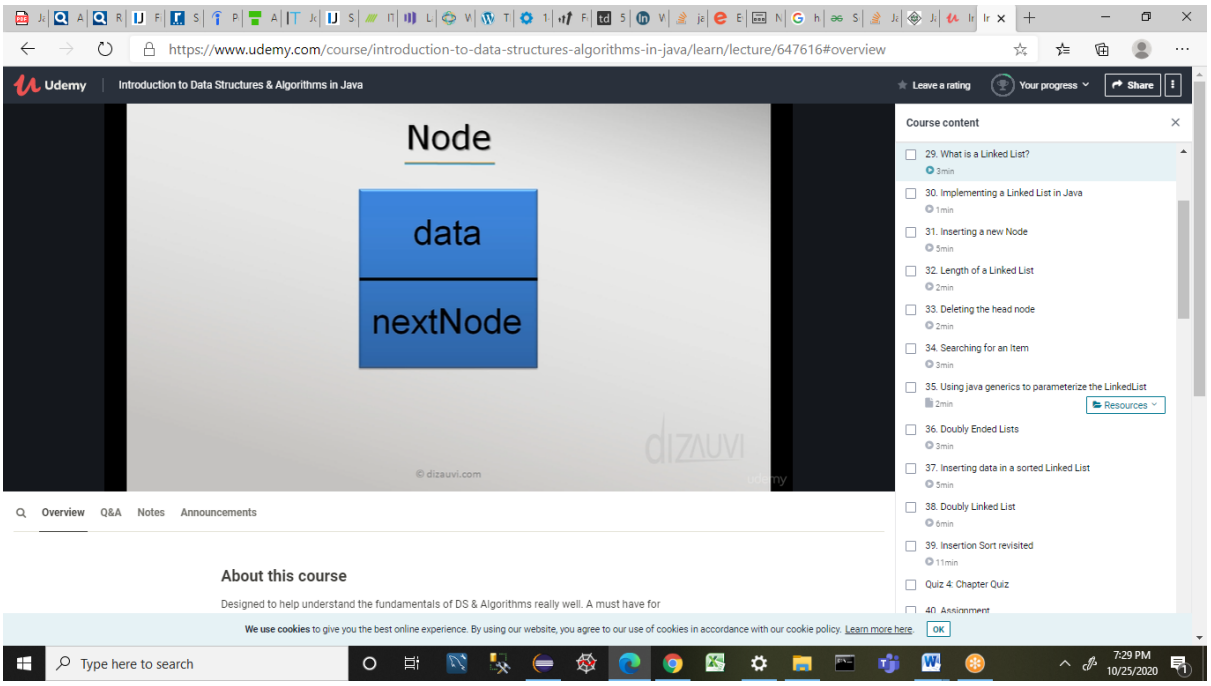
ArrayDeque Methods

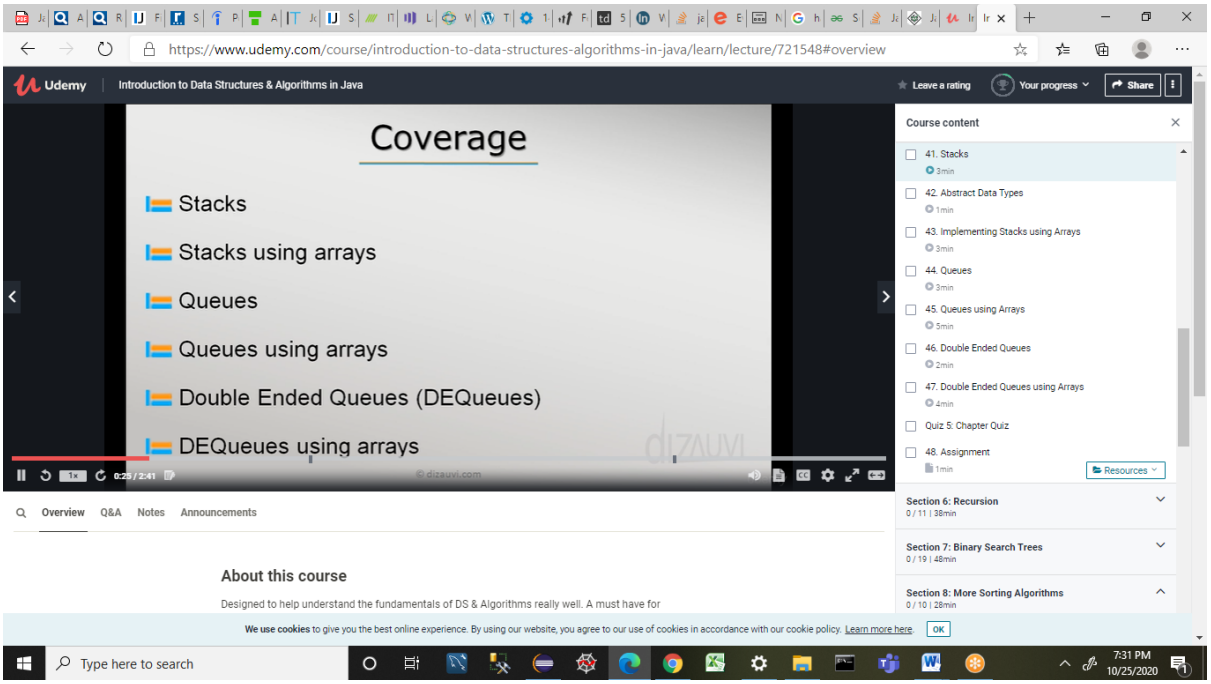
Methods: Add, contains, remove, isempty, size, clear
Iterrator

ArrayBlockingQueue Example

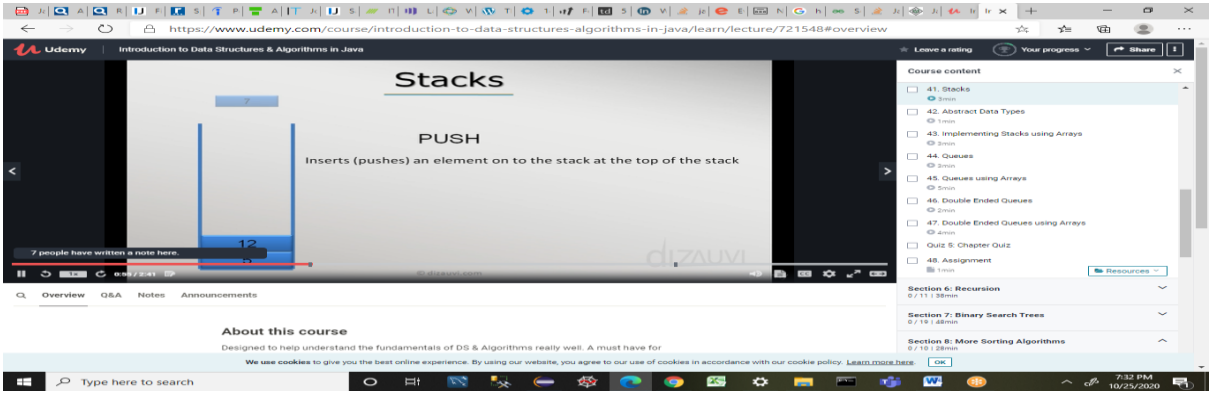
```
Deque<Integer> numbersDeque = new ArrayDeque<>();

	numbersDeque.add(1);
	numbersDeque.add(2);
	numbersDeque.contains(2);
	numbersDeque.remove("Blue");
//		numbersDeque.set(0,1);
//		numbersDeque.get(0);
	numbersDeque.size();
	numbersDeque.clear();
	numbersDeque.isEmpty();
	Iterator<Integer> myIterator2 = numbersDeque.iterator();
	while(myIterator2.hasNext()){
		System.out.println(myIterator.next());
	}
```

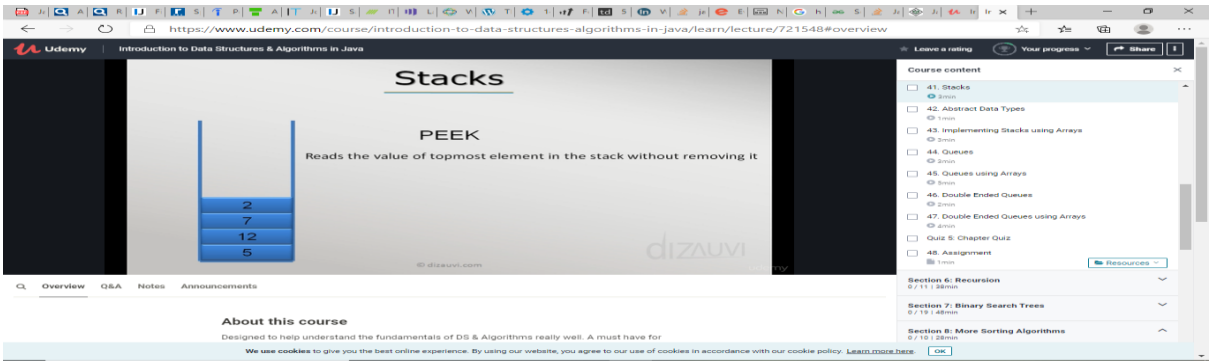


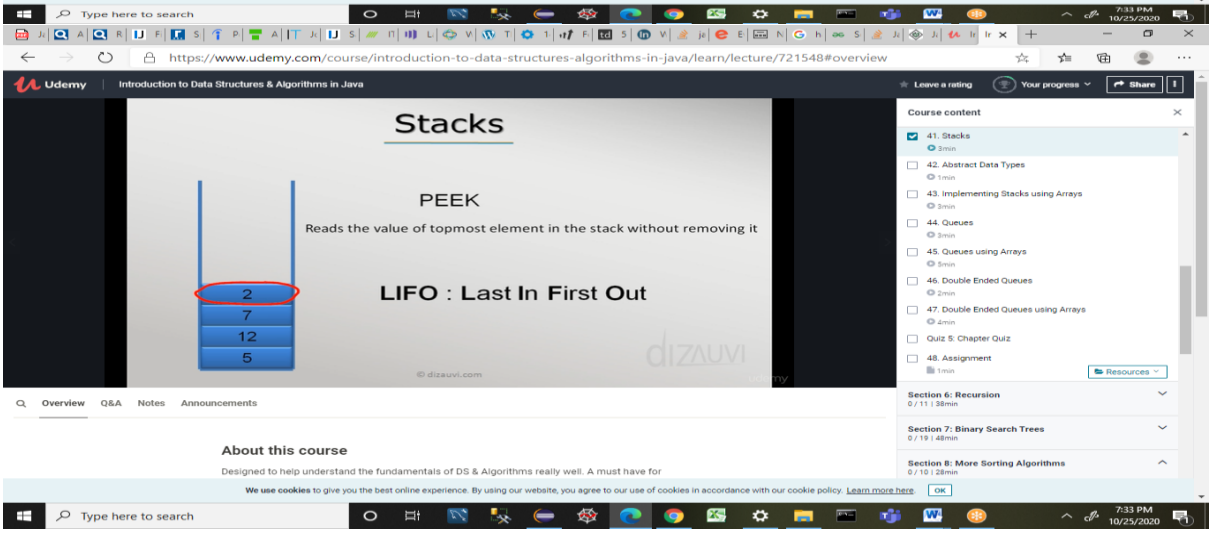


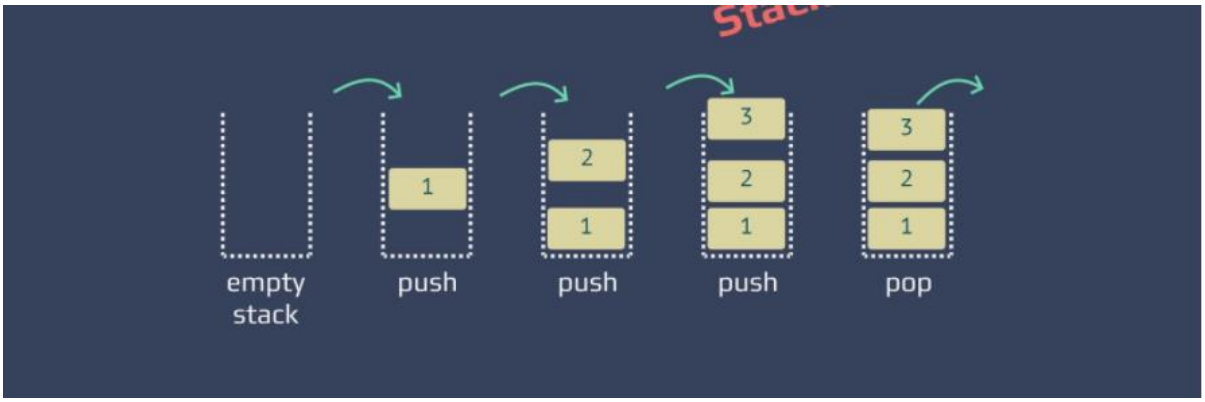


Stack

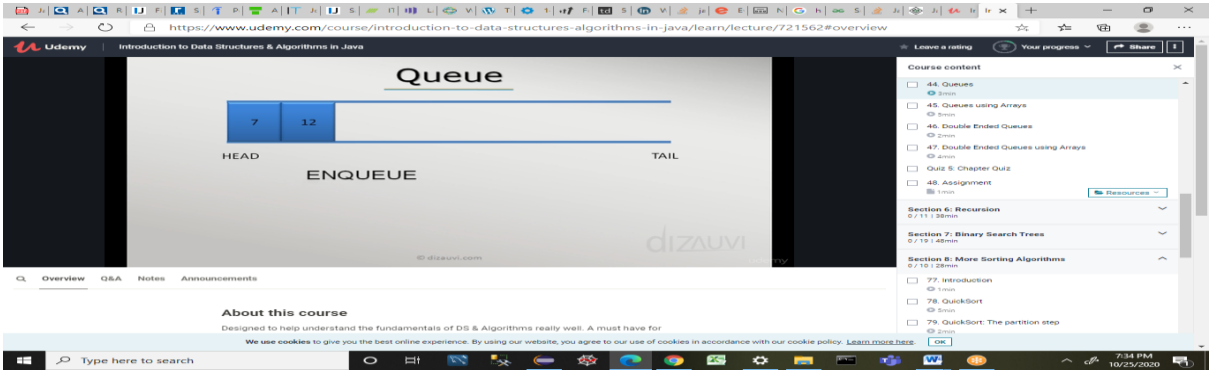


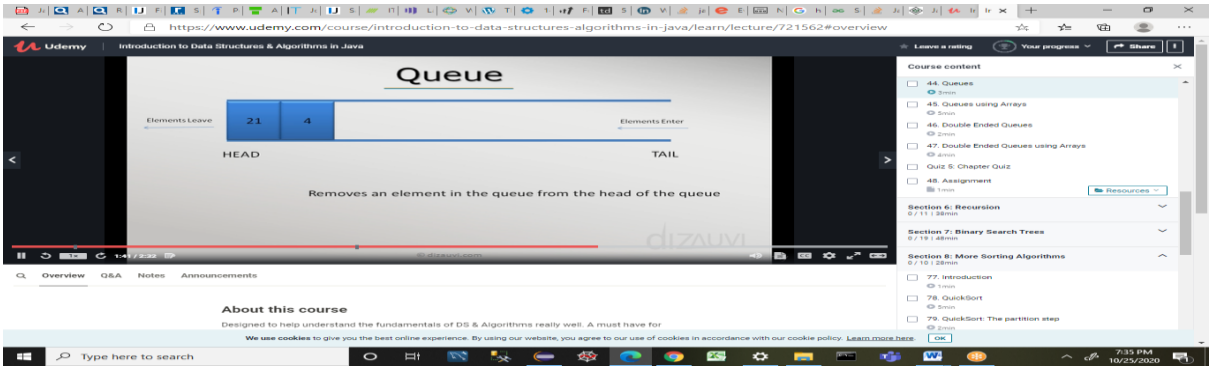


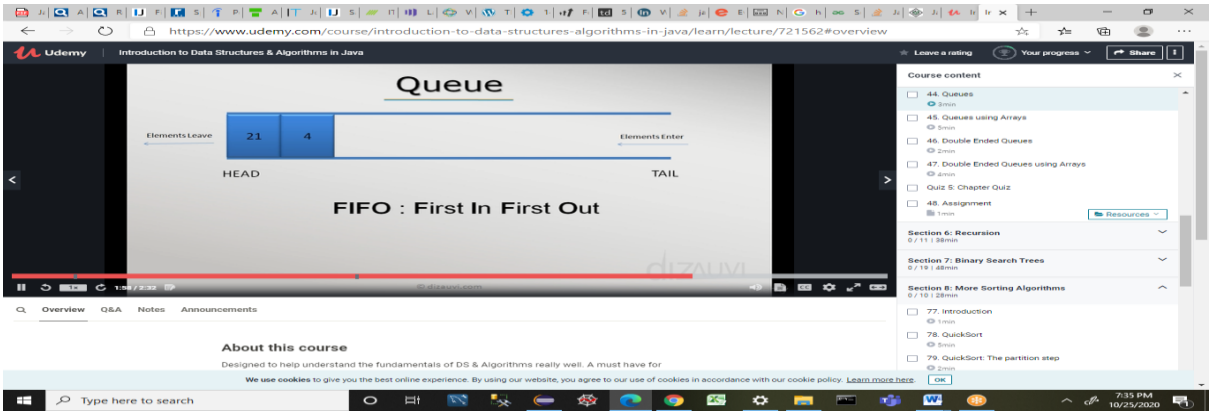




Ques







Enqueue, dequeu and peek and size