Java

Data Types:

Primitive and Non primitive

Primitive:

Type	Byte of memory
byte	1
short	2
int	4
long	8
float	4
double	8
boolean	11
char	1

Non Primitive:

We use Strings, Arrays, etc.

Variables:

Syn:

Access modifier datatype variablename;

Public int a;

Types:

1) Global:

   We can create global variable anywhere inside the class.

   i) Static:
      variable which is declared static is known as static variable
      Static variables can never be local variables

ii) Instance:

The variables which are declared within the class but outside of any method or block or constructor are called “Instance Variables”

Local:

Variables which are declared inside a method or block or constructors such type of variables are called local variables.

Scope of the local variables is exactly same as scope of the block in which we declared.
The only valid modifier for local variables is final.

Operators

there are mainly ‘4’ types of operators. They are

1. Arithmetic operators in Java
2. Relational operators in Java
3. Logical operators in Java
4. Assignment operator in Java

Arithmetic operators in Java

Arithmetic Operators in Java are particularly used for performing arithmetic operations on given data or variables. There are various types of operators in Java, such as

Operators	Operations
+	Addition
-	Subtraction
x	Multiplication
/	Division
%	Modulus

Relational operators in Java

Java relational operators are assigned to check the relationships between two particular operators. There are various relational operators in Java, such as

Operators	Description	Example
==	Is equal to	3 == 5 returns false
!=	Not equal to	3 != 5 returns true
>	Greater than	3 > 5 returns false
<	Less than	3 < 5 returns true
>=	Greater than or equal to	3 >= 5 returns false
<=	Less than or equal to	3 <= 5 returns true

Logical operators in Java

Logical Operators in Java are used for checking whether the expression is true or false. It is generally used for making any decisions in Java programming. Not only that but Jump statements in Java are also used for checking whether the expression is true or false. It is generally used for making any decisions in Java programming.

Operators	Example	Meaning
&& [logical AND]	expression1 && expression2	(true) only if both of the expressions are true
|| [logical OR]	expression1 || expression2	(true) if one of the expressions in true
! [logical NOT]	!expression	(true) if the expression is false and vice-versa

Assignment operator in Java

Assignment Operators are mainly used to assign the values to the variable that is situated in Java programming. There are various assignment operators in Java, such as

Operators	Examples	Equivalent to
=	X = Y;	X = Y;
+=	X += Y;	X = X + Y;
-=	X -= Y;	X = X - Y;
*=	X *= Y;	X = X * Y;
/=	X /= Y;	X = X / Y;
%=	X %= Y;	X = X % Y;

Control Statements

Java provides three types of control flow statements.

1. Decision Making statements
   - if statements
   - switch statement
2. Loop statements
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. Jump statements
   - break statement
   - continue statement

Decision-Making statements:

1) **If Statement:**

a) Simple if statement

b) if-else statement
c) if-else-if ladder
d) Nested if-statement

Simple if statement

Syn:

```
if(condition) {
statement 1; //executes when condition is true
}
```

if-else statement

Syn:

```
if(condition) {
statement 1; //executes when condition is true
}
else{
statement 2; //executes when condition is false
}
```

if-else-if ladder

Syn:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Nested if-statement

Syn:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}
}
```

Switch Statement:

Syn:

```
switch (expression){
   case value1:
    statement1;
    break;
   .
   .
   .
   case valueN:
    statementN;
    break;
   default:
    default statement;
}
```

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied.
  It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

Loop Statements

1. for loop
2. while loop
3. do-while loop

Java for loop

Syn:

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

Java for-each loop

Syn:

```
for(data_type var : array_name/collection_name){
//statements
}
```

Java while loop

Syn:

```
while(condition){
//looping statements
}
```

Java do-while loop

Syn:

```
do
{
//statements
} while (condition);
```

Jump Statements

Java break statement

the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

String, String Buffer and String Builder

Class Name	**Method name**	**Return Type**	
String	length()	int	
String	charAt(int index)	char	
String	equals(String str)	boolean	

String	substring(int begin)	string	
String	concat(String str)	string	
String	toLowerCase()	string	
String	toUpperCase()	string	
String	substring(int begin, int end)	string	
String	indexOf(char ch)	int	
StringBuffer	length()	int	
StringBuffer	capacity()	int	
StringBuffer	charAt(int index)	char	
StringBuffer	indexOf(String str)	int	
StringBuffer	substring(int index)	string	
StringBuilder	length()	int	
StringBuilder	capacity()	int	
StringBuilder	charAt(int index)	char	
StringBuilder	substring(int beginIndex)	string	
StringBuilder	substring(int beginIndex, int endIndex)	string	

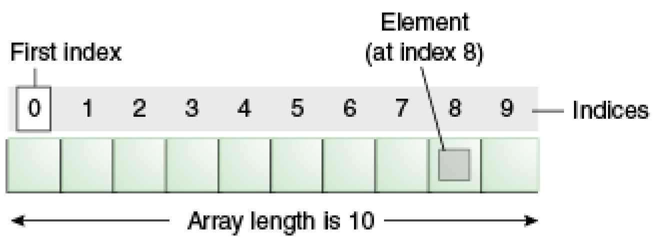
String	StringBuffer	
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

StringBuffer	StringBuilder	
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

```java
package first.welcome;

public class Strings {

	public static void main(String[] args) {
		String s = "java";
		System.out.println(s.charAt(2));
		//This will return index at specific position

		System.out.println(s.length());
		//This will return the length

		String s1 =    s.concat("class");
		System.out.println(s1);
		//This will add the two strings

		System.out.println(s.equals(s1));
		//This will return the boolean value

		System.out.println(s.substring(2));
		//This will return the string starting from the index position given

		System.out.println(s.substring(2, 3));
		//This will return the string from given start and end position

		System.out.println(s.toLowerCase());
		//This will return all char to lower case

		System.out.println(s.toUpperCase());
		//This will return all char to upper case

		System.out.println(s.indexOf("a"));
		//This will return the int value of given char

		Strings x = new Strings();
		x.meth();
		x.meth1();
	}
	void meth() {
		StringBuffer sb = new StringBuffer("Ashraff");
		System.out.println(sb.length());
		//This will return the length of the string buffer

		System.out.println(sb.charAt(2));
		//This will return the char at given index

		System.out.println(sb.capacity());
		//This will return the capacity of the string buffer

		System.out.println(sb.indexOf("r"));
		//This will return the index of given char

		System.out.println(sb.substring(3));
		//This will return the string from specific position
	}
	void meth1() {
		StringBuilder sb1 = new StringBuilder("Basha");

		System.out.println(sb1.length());
		//This will return the length of the string builder

		System.out.println(sb1.capacity());
		//This will return the capacity of the string builder

		System.out.println(sb1.charAt(4));
		//This will return the char at the given index

		System.out.println(sb1.substring(4));
		//This will return the string from given index position

		System.out.println(sb1.substring(1, 4));
		//This will return the string from given index to given last index
	}
}
```

Arrays:

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1. int a[]={33,3,4,5};//declaration, instantiation and initialization
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

Types of Array in java

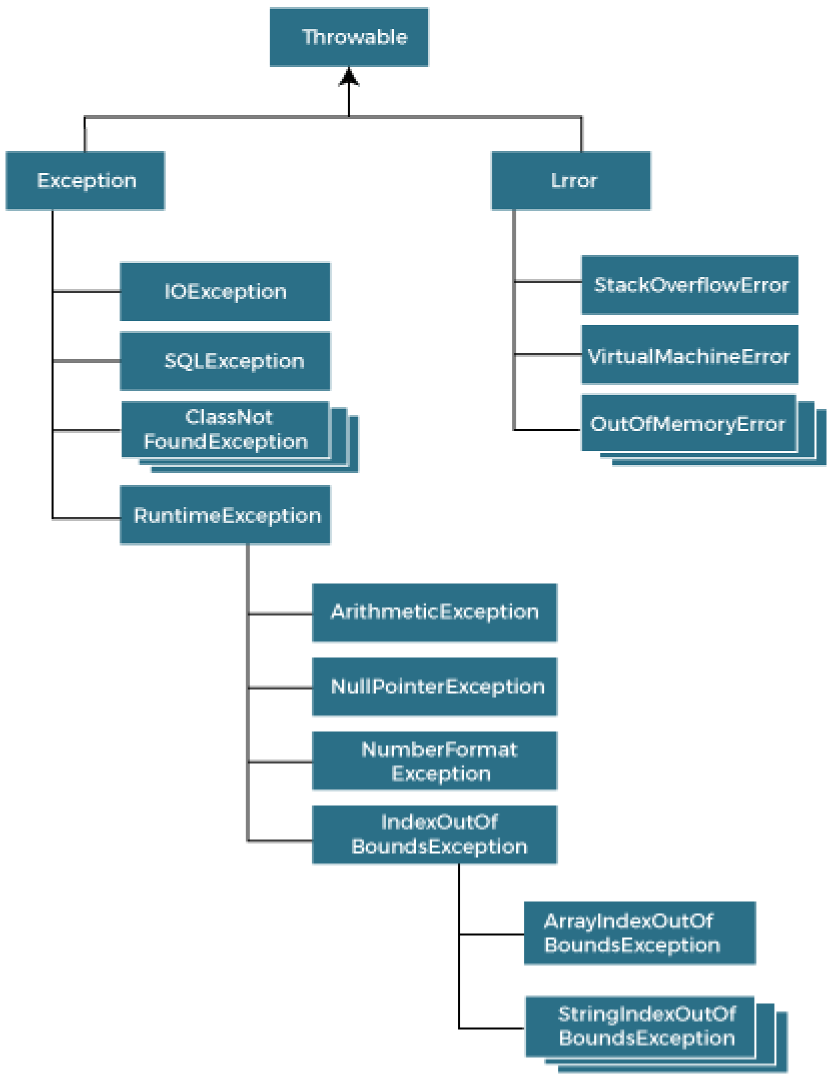
There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Exception Handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Keyword	**Description**
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

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


	int   a[] = {1,2,3}  ;
	System.out.println(a.length-1);

		try {
			System.out.println(a[a.length]);
		}
		catch(Exception e){
			System.out.println(e);
		}
		finally{
			System.out.println(a[0]);
		}
	}
```

OOPS:

There are mainly 4 features of OOPS

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

```
public class Abstraction{

	// Abstraction
	private int a;

	public int getA() {
		return a;
	}

	public void setA(int a) {
		this.a = a;
	}

}
public class Abstraction1 {
	public static void main(String[] args) {

	Abstraction x = new Abstraction();
	x.setA(5);
	System.out.println(x.getA());

	}
}
```

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

```
public abstract class InhrtA {
        public static void main(String[] args) {

        }
        public void meth() {
                System.out.println(2);
        }
        public abstract void meth1() ;



}
public class InhrtB extends InhrtA{
        public static void main(String[] args) {
                InhrtB x = new InhrtB();
                x.meth1();
        }
        public void meth1(){
        System.out.println(2);
                }
        }
```

Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

```
public interface IntrfA {
public static void main(String[] args) {

}
                    public void meth() ;

                    }
public interface IntrfB {
          public static void main(String[] args) {

          }
          void meth();
}
public class InhrtC implements IntrfA ,IntrfB {
          public static void main(String[] args) {
                    InhrtC x = new InhrtC();
                    x.meth();
          }
          public void meth() {
                    System.out.println(2);
          }
}
```

Polymorphism in Java

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**

```
package oops;

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

	}
	public int meth() {
		return 0;
		//no parameter
	}
	public int meth(int a) {
		return 1;
		//single parameter
	}
	public int meth(int a, int b) {
		return 2;
		// multi parameter
	}
	public int meth(int a, String b) {
		return 3;
	}
	public int meth(String a, int b) {
		return 4;
		//By changing the order of parameters
	}


}
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**

```
package oops;

public class Method_ovrrid {
	public int meth() {
		return 2;
}
	public static void main(String[] args) {

	}
	}
public class Method_ovrrid1 extends Method_ovrrid {

	public static void main(String[] args) {

		Method_ovrrid1 x = new Method_ovrrid1();
		int a = x.meth();
		System.out.println(a);
	}
}
```

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

```
public abstract class InhrtA {
	public static void main(String[] args) {

	}
	public void meth() {
		System.out.println(2);
	}
	public abstract void meth1() ;
}

public class InhrtB extends InhrtA{
	public static void main(String[] args) {
		InhrtB x = new InhrtB();
		x.meth1();
	}
	public void meth1(){
	System.out.println(2);
		}
	}
```

multiple inheritance through interface

```
public interface IntrfA {
public static void main(String[] args) {

}
		public void meth() ;

		}
public interface IntrfB {
	public static void main(String[] args) {

	}
	void meth();
}
public class InhrtC implements IntrfA ,IntrfB {
	public static void main(String[] args) {
		InhrtC x = new InhrtC();
		x.meth();
	}
	public void meth() {
		System.out.println(2);
	}
}
```

Java Threads

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Creating a Thread

There are two ways to create a thread.

It can be created by extending the Thread class and overriding its run() method:

Another way to create a thread is to implement the Runnable interface:

Running Threads

If the class extends the Thread class, the thread can be run by creating an instance of the class and call its start() method:

```
package first.welcome;

public class Threads_ extends Thread {

        public static void main(String[] args) {
              Threads_ thread = new Threads_();
              thread.start();
              System.out.println("This code is outside of the thread");
             }
             public void run() {
              System.out.println("This code is running in a thread");
             }
            }
```

Collections in Java

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).

framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

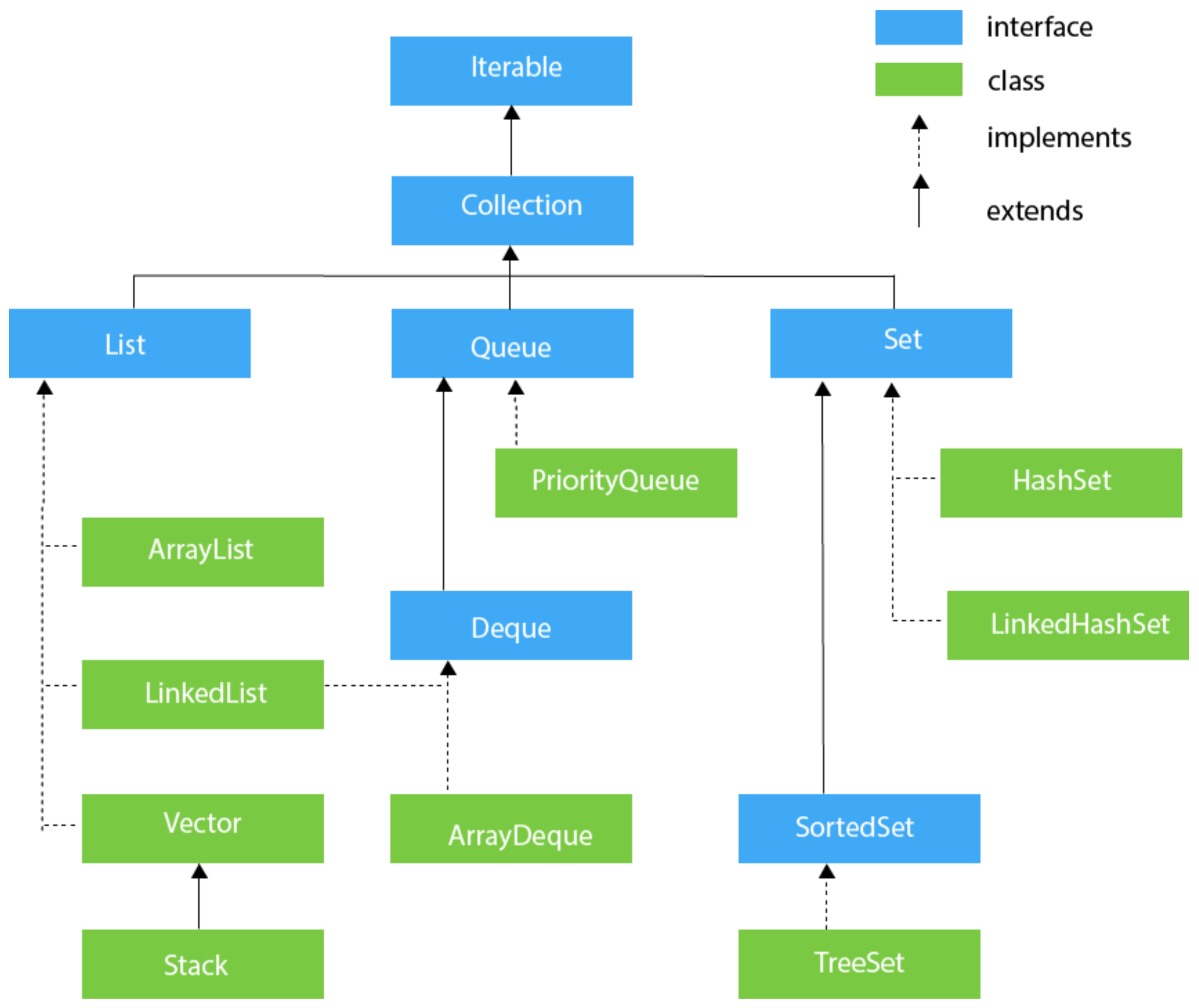
Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.

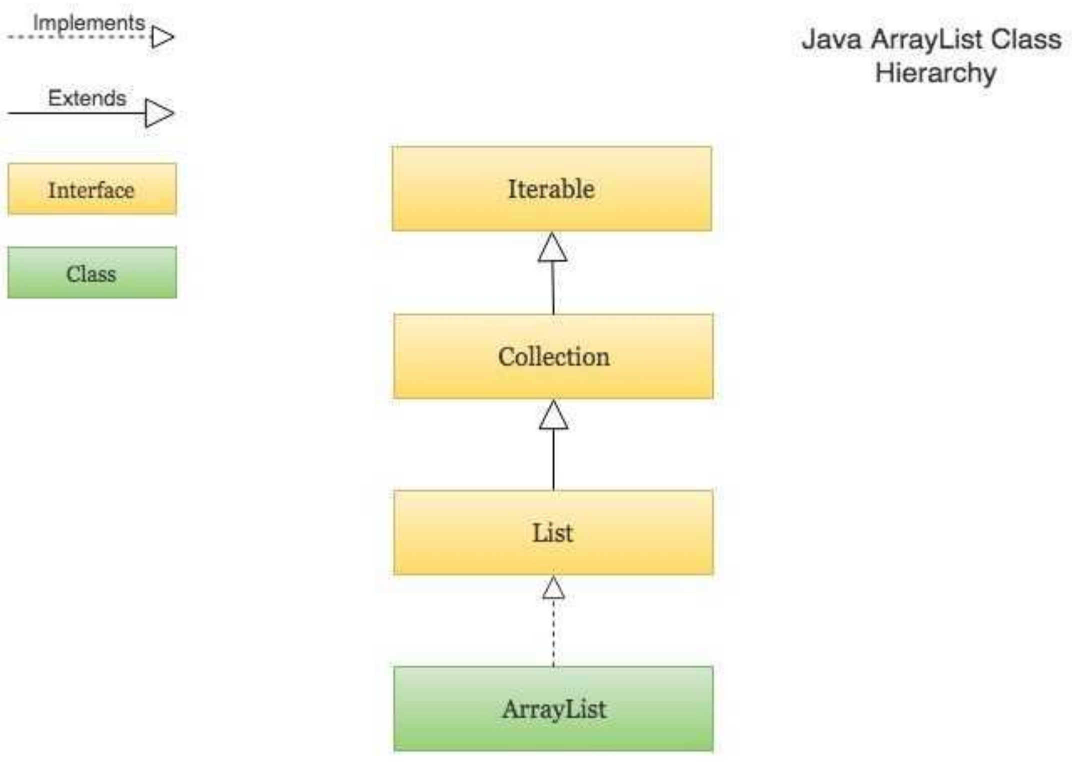


Java ArrayList

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases.

- Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

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

		List <String> a = new ArrayList<>();
		a.add("hi");
		a.add("java");
		a.add("class");
		//System.out.println(a);

		List <String> b = new ArrayList<>(a);

		List <String> c = new ArrayList<>();
		c.add("Training");
		c.add("session");

		b.addAll(c);
		System.out.println(b);
	Collections.sort(b);
	System.out.println(b);
	}
}
```



Java LinkedList class

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

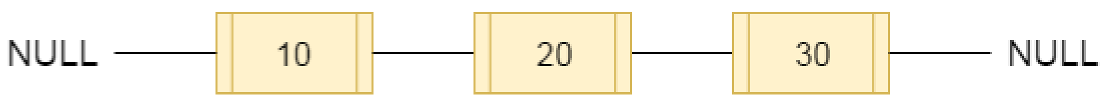
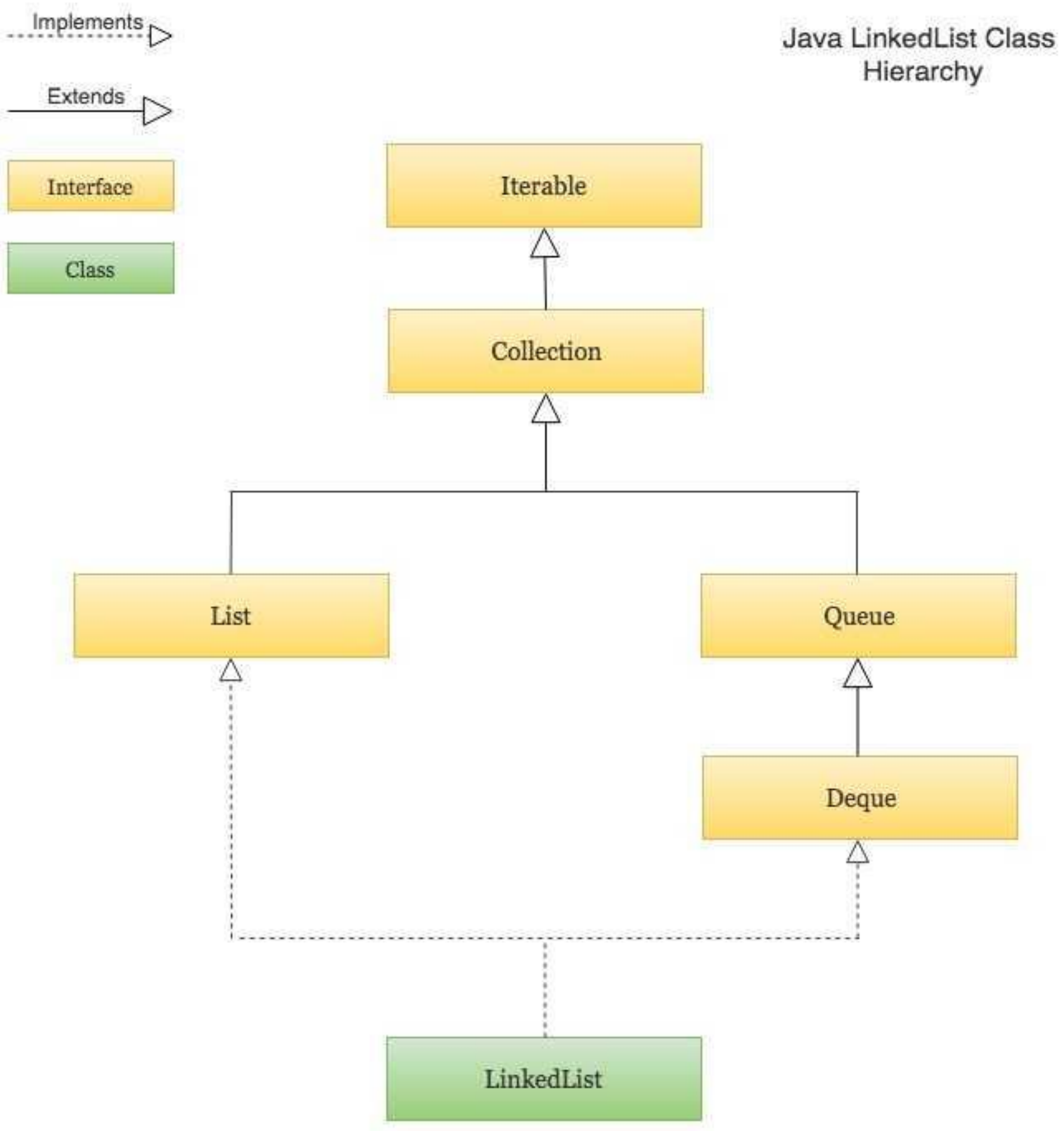


fig- doubly linked list

```
package Colleccions;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class LinkedList_c {
        public static void main(String[] args) {
                LinkedList <String> friends = new LinkedList<>();
                 // Adding new elements to the end of the LinkedList using add()
method.
      friends.add("Rajeev");
      friends.add("John");
      friends.add("David");
      friends.add("Chris");

      System.out.println("Initial LinkedList : " + friends);

      // Adding an element at the specified position in the LinkedList
      friends.add(3, "Lisa");
      System.out.println("After add(3, \"Lisa\") : " + friends);

      // Adding an element at the beginning of the LinkedList
      friends.addFirst("Steve");
      System.out.println("After addFirst(\"Steve\") : " + friends);

      // Adding an element at the end of the LinkedList (This method is
equivalent to the add() method)
      friends.addLast("Jennifer");
      System.out.println("After addLast(\"Jennifer\") : " + friends);

      // Adding all the elements from an existing collection to the end of the
LinkedList
      List<String> familyFriends = new ArrayList<>();
      familyFriends.add("Jesse");
      familyFriends.add("Walt");

      friends.addAll(familyFriends);
      System.out.println("After addAll(familyFriends) : " + friends);
   }
```

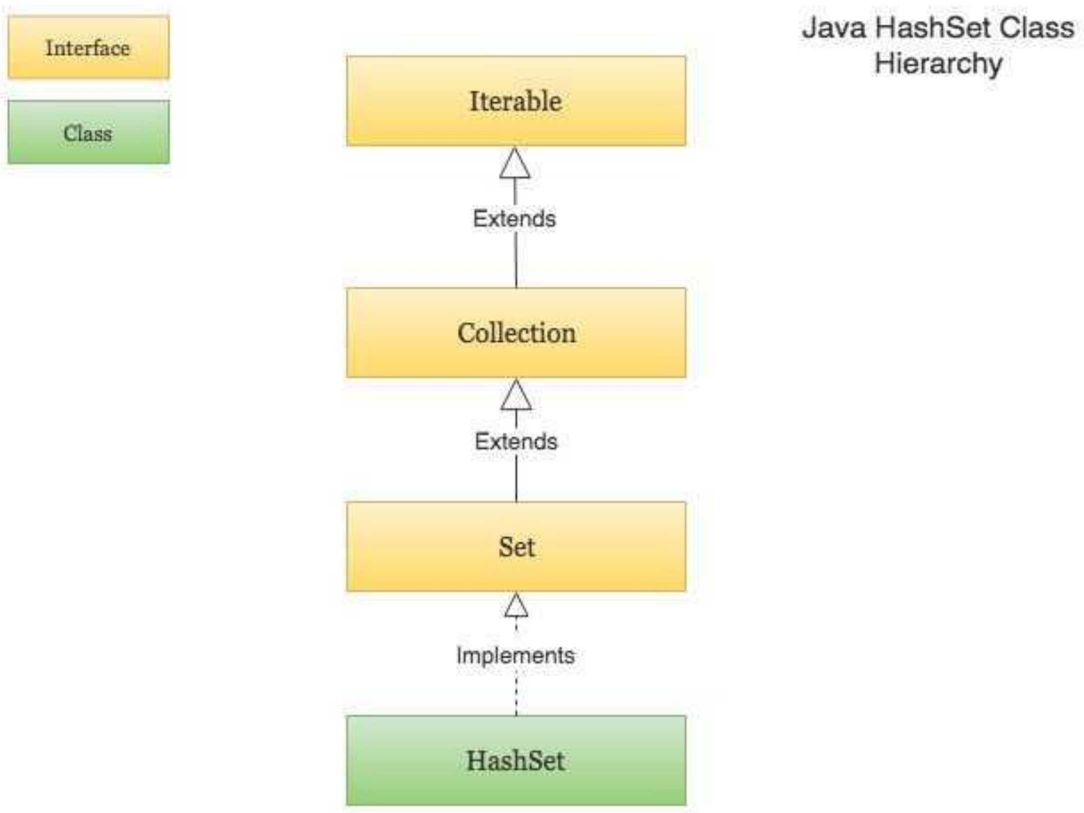
}



Java HashSet

- HashSet stores the elements by using a mechanism called **hashing.**

- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

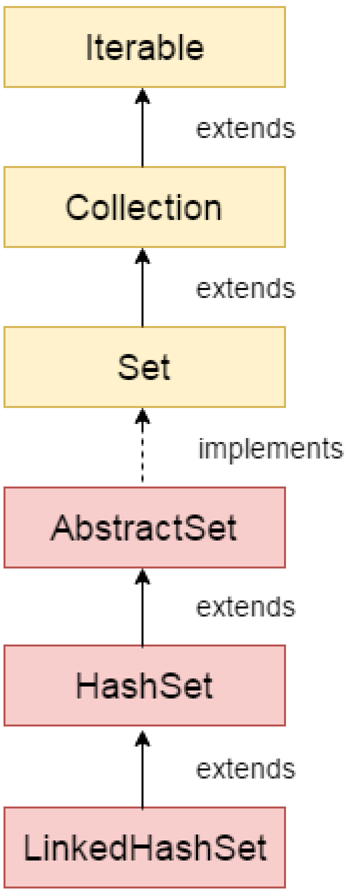


```
package Collecctions;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;
public class HashSet_c {
	public static void main(String[] args) {
		Set <String> Hs= new HashSet<>();
		Hs.add("Monday");
		Hs.add("Wednesday");
		Hs.add("Tuesday");
		Hs.add("Thursday");
```

```
                Hs.add("Saturday");
                System.out.println(Hs);
                //Collections.sort(Hs);

        }

}
```

Java LinkedHashSet Class

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operations and permits null elements.
- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.



```
public class Book {
	int id;
	String name;
	int quantity;
	public Book(int id, String name, int quantity) {
		this.id=id;
		this.name=name;
		this.quantity=quantity;
	}

}
package Collecctions;
import java.util.Set;
import java.util.LinkedHashSet;

public class LinkedHashSet_c  {
	public static void main(String[] args) {
		LinkedHashSet<Book> Hls= new LinkedHashSet<>();
		Book b1 = new Book(101, "Silent Patient", 28);
		Book b2 = new Book(102,"Alchemist", 07);
		Book b3 = new Book(103,"The Maidens", 10);

		Hls.add(b1);
		Hls.add(b2);
		Hls.add(b3);

		for(Book b: Hls) {
			System.out.println(b.id +" " +b.name +" "+ b.quantity);

		}
	}


		}
```

Java TreeSet

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quite fast.

- Java TreeSet class doesn't allow null elements.
- Java TreeSet class is non-synchronized.
- Java TreeSet class maintains ascending order.
- The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

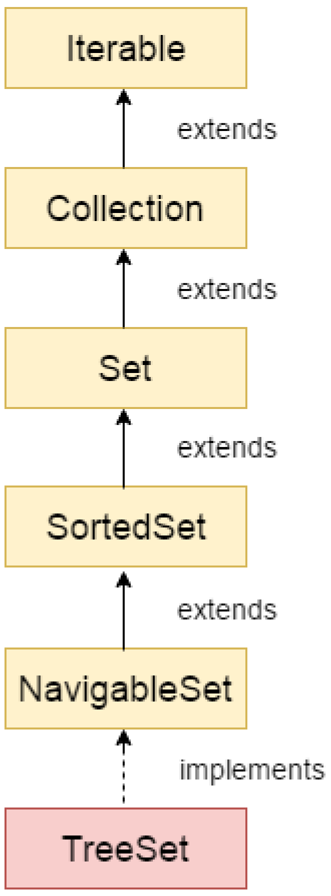
Internal Working of The TreeSet Class

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

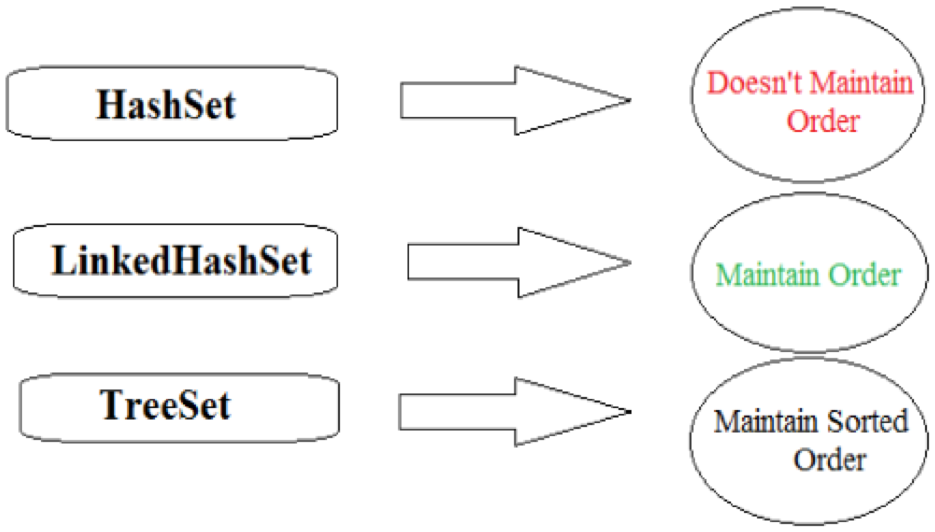
Synchronization of The TreeSet Class

As already mentioned above, the TreeSet class is not synchronized. It means if more than one thread concurrently accesses a tree set, and one of the accessing threads modify it, then the synchronization must be done manually. It is usually done by doing some object synchronization that encapsulates the set. However, in the case where no such object is found, then the set must be wrapped with the help of the Collections.synchronizedSet() method. It is advised to use the method during creation time in order to avoid the unsynchronized access of the set. The following code snippet shows the same.

```
TreeSet treeSet = new TreeSet();
Set syncrSet = Collections.synchronziedSet(treeSet);
```



All the three class doesn’t accept duplicates elements.

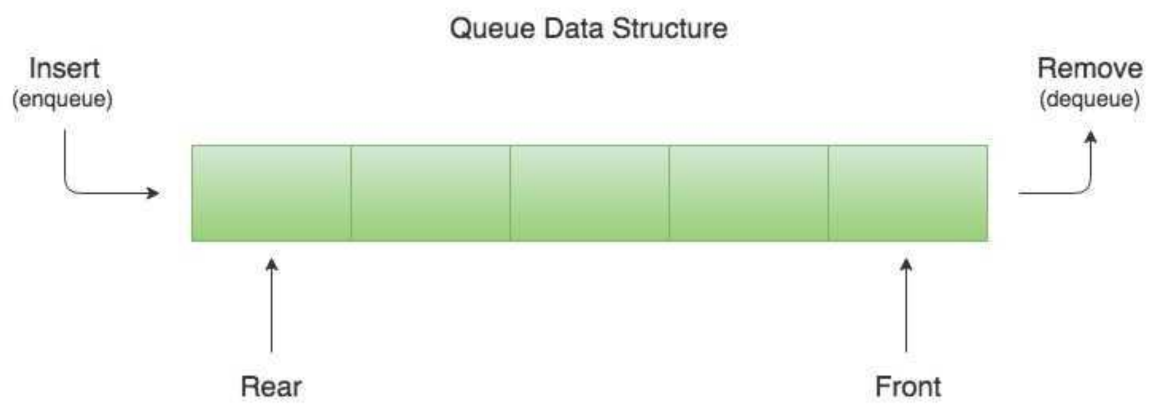


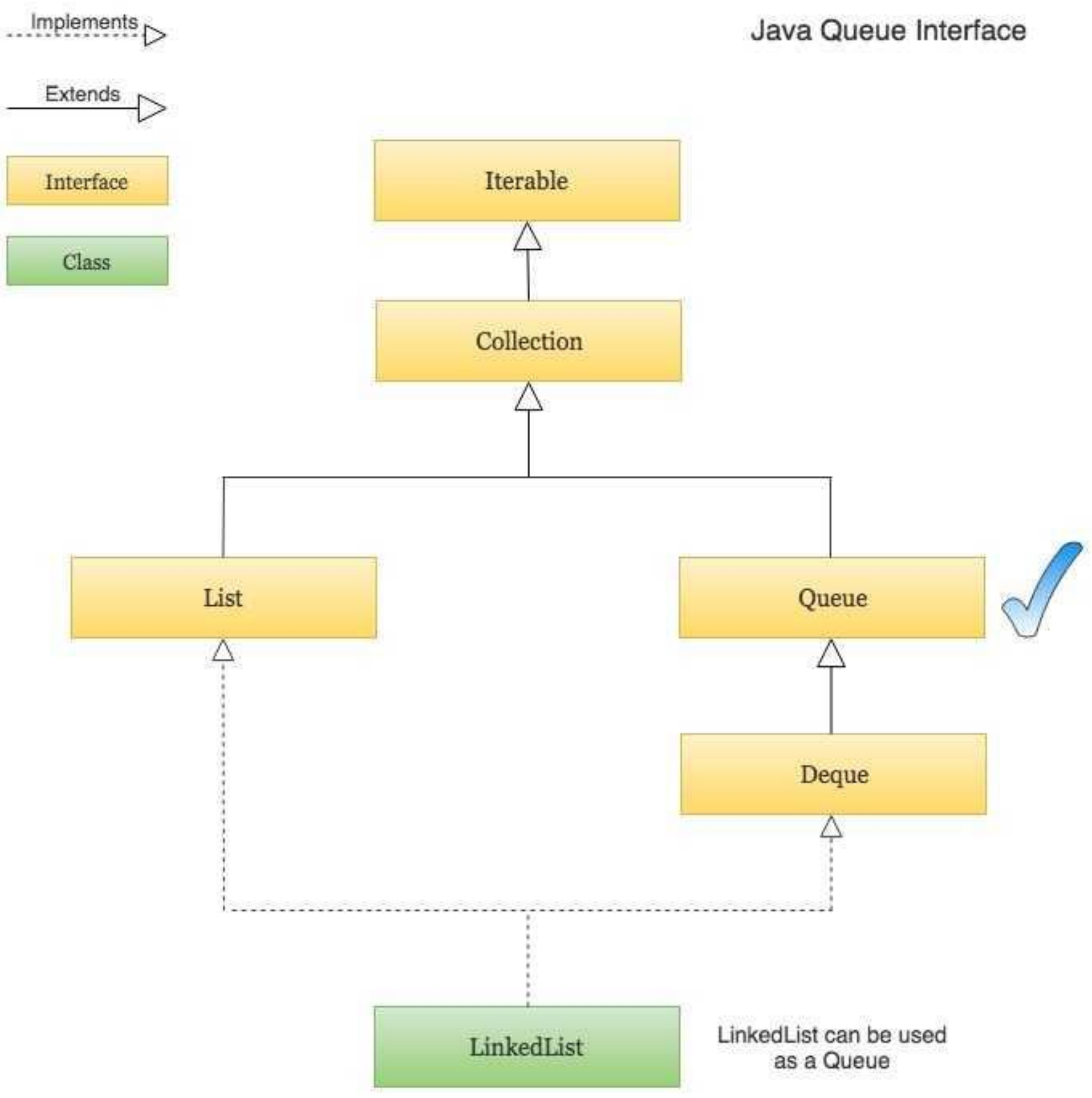
package Collecctions;

```
import java.util.SortedSet;

import java.util.Comparator;

import java.util.Set;

import java.util.TreeSet;


public class Treeset_c {

	public static void main(String[] args) {


	SortedSet <String> Ts= new
TreeSet<>(String.CASE_INSENSITIVE_ORDER);

	Ts.add("Apple");

	Ts.add("Orange");

	Ts.add("Grapes");

	Ts.add("Banana");

	System.out.println(Ts);

	Treeset_c S= new Treeset_c();

	S.meth1();


	}


	void meth1() {

		SortedSet<String> Ts1= new
TreeSet<>(Comparator.reverseOrder());
```

```
                    Ts1.add("Wednesday");

                    Ts1.add("Friday");

                    Ts1.add("Monday");

                    Ts1.add("Sunday");

                    System.out.println(Ts1);

          }

}
```

Java Queue Interface



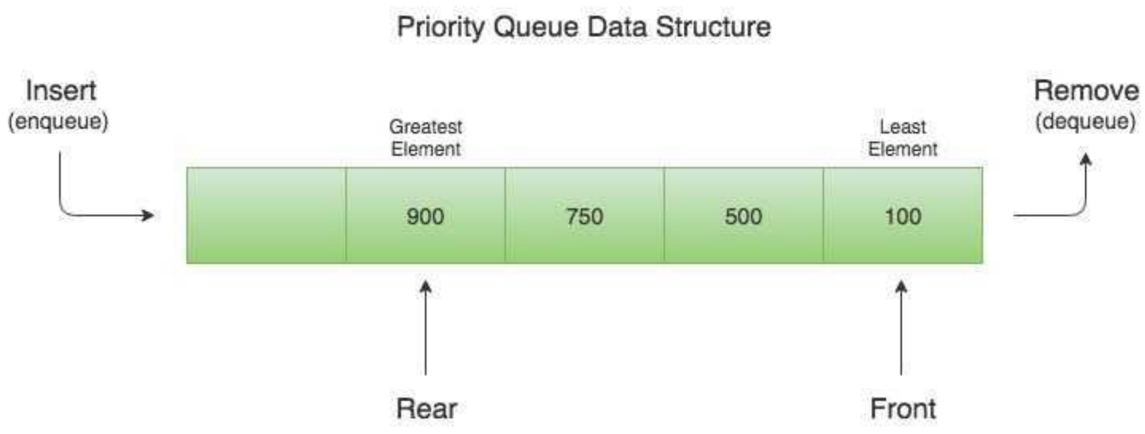


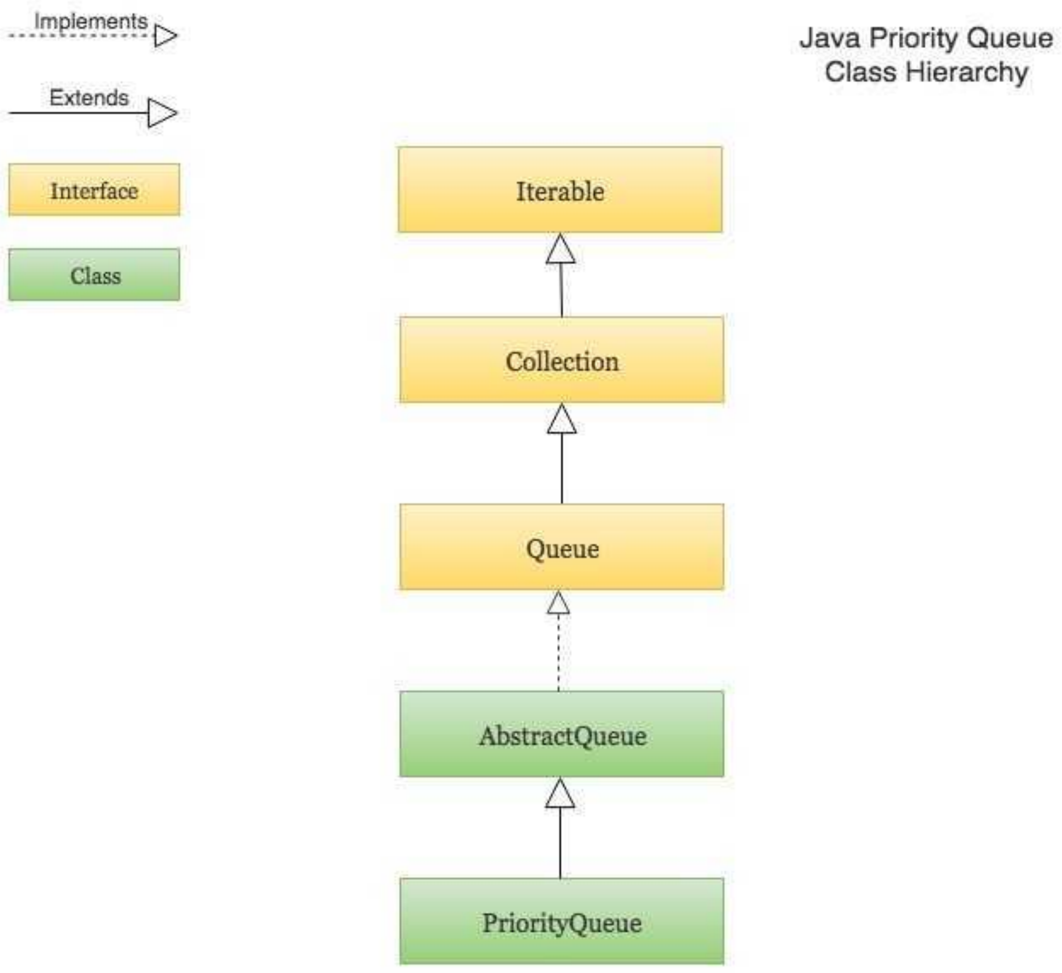
- As discussed earlier, FIFO concept is used for insertion and deletion of elements from a queue.
- The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.
- PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.
- The NullPointerException is raised, if any null operation is done on the BlockingQueues.
- Those Queues that are present in the *util* package are known as Unbounded Queues.
- Those Queues that are present in the *util.concurrent* package are known as bounded Queues.

- All Queues barring the Deques facilitates removal and insertion at the head and tail of the queue; respectively. In fact, deques support element insertion and removal at both ends.

```
package Collecctions;

import java.util.Queue;

import java.util.LinkedList;

import java.util.Iterator;




public class Queue_c {

        public static void main(String[] args) {

                Queue <String> Q= new LinkedList<>();

                Q.add("Java");

                Q.add("Training");

                Q.add("Session");


                Iterator <String> I=  Q.iterator();

                while(I.hasNext()) {

                        System.out.println(I.next());

                        //System.out.println(Q);


                }

                }
}
```

PriorityQueue Class





A priority queue in Java is a special type of queue wherein all the elements are **ordered** as per their natural ordering or based on a custom `Comparator` supplied at the time of creation.

The *front* of the priority queue contains the least element according to the specified ordering, and the *rear* of the priority queue contains the greatest element.

```
package Collecctions;

import java.util.PriorityQueue;

import java.util.Comparator;


public class PriorityQ_c {

	public static void main(String[] args) {

		Comparator <String> C = new Comparator<String>() {

			@Override

	 public int compare(String s1, String s2) {

	    return s1.length() - s2.length();

	 }

		};


	PriorityQueue <String> Pq = new PriorityQueue<>(C);

	//Enque

	Pq.add("Shaik");

	Pq.add("Mohammed");

	Pq.add("Ashraff");
```

```
        Pq.add("Basha");

        System.out.println(Pq);


        //Dequeue

        while(!Pq.isEmpty()) {

                System.out.println(Pq.remove());

        }

        }



    }
```