Java Generics and Collections with Generics

Overview

- Java Generics
- Collections with Generics

Java Generics

- Starting with JDK 1.5 in Java are added generics like C++ templates. Generics are based on the *parametric polymorphism* found in functional languages as ML and Haskell.
- Java generics are implemented by **"Erasure" because** *List <Integer> , List <String>,* etc. are represented at execution by the same type, *List.*
- The *cast-mechanism* is realized implicit at generics.
- It is named, *cast-iron guarantee*, because we have no problems and is realized implicit.

- Generics are basically the parameterization of types.

- Creating classes, interfaces, generalized methods was allowed in Java by using type references *Object* (by upcasting), *Object* class being a superclass of all other Java classes.
- So a **reference of type** *Object* may refer any object type. This usually is not „type safety”.

- **Generics** added this „type safety” and it is not necessary to use an explicit cast mechanism to realize the translation between *Object* and the effective working type.
- With generics all *cast*-s are **automatically and implicit** realized.

- The implementation of generics via ***Erasure*** brings benefits such as:

-the things are simple, they do not add something new in the end

-things remain small, being a single implementation for *List* (in our case example), not a new version for every generic type

-allows the evolution of the same library to be used generically or not (with some restrictions).

- Practically the byte-code obtained with generic and non-generic is the same.

- **Constructions:**

-*new String[size],* array of *String*-s, will store in the array an indication that we use *String*-s

-***new ArrayList <String>***, will allocate a list, but does not memorize an item's indication about the nature of the items, *String*, that's because from ***Java 7*** is equivalent with ***new ArrayList <>*****.**

- **Generics in Java are deliberately restrictive to keep them simple and easy to understand.**

Generic methods

- Generic methods are methods that introduce their own type parameters. This is like declaring a generic type, but the type parameter's scope is limited to the method where it is declared.
- *Static* and *non-static generic methods* are allowed, as well as generic class *constructors*.

- The **syntax for a generic method** includes a list of type parameters, inside angle brackets, which appears before the method's return type. For **static generic methods**, the type parameter section must appear before the method's return type.
- The *Util* class includes a generic method, *compare( )*, which compares two *Pair* objects:

```
public class Util {
  public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
              p1.getValue().equals(p2.getValue());  }
}
```

Pair is a class that allows collection management based on *Key –Value*.

The syntax for invoking the *compare()* method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.<Integer, String>compare(p1, p2);
```

Generic Classes

- A generic class is a special type of class that associates one or more non-specified Java types upon instantiation.
- Here is an example of how to initialize a generic class with 2 Java types:

public class Example<Type1, Type2>{

private Type1 t1;

private Type2 t2;…

}

//You may include methods that set each of the types to the types that are initiated when a new object of this class is created.

Type Interface Diamond < >

A *type interface diamond* enables you to create a generic method as you would an ordinary method, without specifying a type between angle brackets.

- *<Type1, Type2>* is the type interface diamond for *Example* on the previous slide.

Why a diamond?

- Because the angle brackets can be referred to as the diamond < >.
- Typically if there is only 1 type inside the diamond we use *<T>* where *T* stands for *Type*.

When working with generic types, remember the following:

• The types must be ***identified at the instantiation*** of the class.

• Your class should contain methods that set the types inside the class to the types passed into the class upon creating an object of the class.

One way to look at generic classes is by understanding what is happening behind the code.

```
public class Example<Type1, Type2>{
private Type1 t1;
private Type2 t2;
…      }
```

This code can be interpreted as a class that creates 2 reference objects, *t1* and *t2* of *Type1* and *Type2.*

Type1 and *Type2* are not the type of objects required to be passed in upon initializing an object *Example,* they are simply placeholders, or variable names, for the actual type that is to be passed in.

Using these placeholders allows for the class to include **any Java Wrapper** type, or **other class** type; they become whatever type is initialized with the object creation.

Inside of the generic class, when you create an object of *Type1* or *Type2*, you are actually creating objects of the types initialized when an *Example* object is created.

How to initialize objects

Example<String, Integer> stringInt = new Example<String, Integer>("Gen",10);

The only difference between creating an object from a regular class versus a generics class is *<String, Integer>*. This is how to tell the *Example* class what type of objects you are using with that particular object.

In other words, *Type1 is* a *String* type, and *Type2* is an *Integer* type. The benefit to having a generic class is that you can identify multiple objects of type *Example* with different types given for each one, so if we wanted to, we could initialize another object *Example* with *<Double, String>:*

Example<Double, Integer> doubleInt = new Example< >(7.7, 7); //>Java 7

Examples with generics

- In the first example the compiler does not create different versions of *Gen* or another generic class as in C ++.
- It removes all generic information, replacing all the required casts.
- Finally will be a single version of *Gen*, and the process of eliminating generic information is called "erasure."

Generics with a generic type T

```
//G1
class Gen <T> {
	T ob;
	//cons
	Gen (T o){
		ob=o;
	}
	//get
	T getOb(){
		return ob;
	}
	//show
	void ShowType(){
		System.out.println("Type of T is "+
ob.getClass().getName());
	}
}//Gen
```

```
public class GenDemo {
        public static void main(String[ ] args) {
Gen <Integer> iob;//important !!!– the type of iob is Integer
iob= new Gen <Integer> (88);//explicit type
iob.ShowType();
int v= iob.getOb();
System.out.println("value v= " + v);
iob= new Gen <> (77);//implicit type (no type specified)
iob.ShowType();
 v= iob.getOb();
System.out.println("value v= " + v);
System.out.println();

Gen <String> strOb= new Gen <> ("Generics simple test");
strOb.ShowType();
String str = strOb.getOb();
System.out.println("value str= " + str);
        }//main
}//GenDemo
```

Two generics

```
//G2
class TwoGen <T, V>{
        T ob1;
        V ob2;
        //cons
        TwoGen(T o1, V o2){
                ob1=o1;
                ob2=o2;
        }
        void showTypes(){
System.out.println("Type of T is "+ ob1.getClass().getName());
System.out.println("Type of V is "+ ob2.getClass().getName());
                }
        T getOb1(){
                return ob1;
        }
        V getOb2(){
                return ob2;
        }
}//TwoGen
```

```
public class SimpTwoGen {
        public static void main(String[ ] args) {
        TwoGen <Integer, String> tgOb = new TwoGen<Integer,
String> (88, "Two Generics ");//explicit
        tgOb.showTypes();
        int v = tgOb.getOb1();
        System.out.println(" value is = "+v);
        String str = tgOb.getOb2();
        System.out.println(" value is = "+str);
        TwoGen <Integer, String> tgOb2 = new TwoGen<> (77,
"Other Two Generics ");//implicit
        tgOb2.showTypes();
        v = tgOb2.getOb1();
        System.out.println(" value is = "+v);
        str = tgOb2.getOb2();
        System.out.println(" value is = "+str);
        }//main
}//SimpTwoGen
```

Bounded Generics

- Sometimes it is intended to limit, restrict or ***specialize*** the type of data, such as only the numeric one. We can not assume that type *T* is only numeric and that is why we will extend the working class using a template mechanism for the type in the base superclass that can be *Number*.
- The general form will be:

class Work <T extends Superclass> {…}

- This mechanism is called bounded types, ***specialization*** – specification - limitation, and *Superclass* is an *upperbound*, *T* being *specialized* in this type.

Using a date type as a Java specialization, one or more class interfaces can be added. In this case the generic type *T* will be *specialized* and then using the **&** operator will continue to add interfaces.

class Gen <T extends MyClass & MyInterface>{…}

```
//G3
class Stats <T extends Number>{
        T [ ] nums;
        Stats(T[ ] a){
                nums=a;
        }
        double average(){
                double sum=0.0;
                for(int i=0;i<nums.length;i++)
                        sum+=nums[i].doubleValue();
                return sum/nums.length;
        }
}//Stats
```

```
public class BoundsDemo {
        public static void main(String[ ] args) {
        Integer inums[ ] = {1,2,3,4,5};
        //Stats<Integer> iob = new Stats<Integer>(inums);
        Stats<Integer> iob = new Stats<>(inums);
        double v= iob.average();
        System.out.println("iob average is= "+v);

        Double dnums[ ] = {1.1,2.2,3.3,4.4,5.5};
        Stats<Double> dob = new Stats<>(dnums);
        double w= dob.average();
        System.out.println("dob average is= "+w);

        //cu String nu e permis fiind bounded la Number
        }//main
}//BoundsDemo
```

Wildcards as generic parameters

- We may **want to use a method in a generic class that accepts as a parameter** a type of data to be the ***object of the generic class***, and thus possible with different types of values (let us say numeric type *Integer, Float, Double*).
- So in Java the generic method instead of the generic type *T* will specify ***?* (*wildcard*)** as a parameter.

- In general, to set an *upperbound* for a *wildcard* it will use the expression:

<? extends Superclass>, where *Superclass* is the name of the class that serves as *upperbound*.

- You can also use *lowerbound* for *wildcards* as follows:

<? super Subclass>

- In this case, only classes that are *Superclass-es* of *Subclass* are accepted as arguments. This is an exclusive clause because it will not fit the class with the *Subclass*.

Wildcard generics

```
//G4
class StatsW <T extends Number>{
        T [ ] nums;
        StatsW(T[ ] a){
                nums=a;
        }
        double average(){
                double sum=0.0;
                for(int i=0;i<nums.length;i++)
                        sum+=nums[i].doubleValue();
                return sum/nums.length;
        }
        boolean sameAvg(StatsW<?> ob){
                if (average() == ob.average())return true;
                return false;
        }
}//StatsW
```

```
public class WildcardDemo {
        public static void main(String[ ] args) {
        Integer inums[ ] = {1,2,3,4,5};
        StatsW<Integer> iob = new StatsW<>(inums);
                double v= iob.average();
                System.out.println("iob average is= "+v);
        Double dnums[ ] = {1.1,2.2,3.3,4.4,5.5};
        StatsW<Double> dob = new StatsW<>(dnums);
                double w= dob.average();
                System.out.println("dob average is= "+w);
        Float fnums[ ] = {1.0f,2.0f,3.0f,4.0f,5.0f};
        StatsW<Float> fob = new StatsW<>(fnums);
                double x= fob.average();
                System.out.println("fob average is= "+x);
        System.out.println("Average of iob and dob");
                if(iob.sameAvg(dob))System.out.println(" are the same ");
                        else System.out.println(" difer ");
        System.out.println("Average of iob and fob");
                if(iob.sameAvg(fob))System.out.println(" are the same ");
                        else System.out.println(" difer ");
                }//main
}//WildcardDemo
```

Bounded wildcard

```
//G5
class TwoD{
		int x, y;
		TwoD(int a, int b){
				x=a;
				y=b;
		}
}

class ThreeD extends TwoD{
		int z;
		ThreeD(int a, int b, int c){
				super(a,b);
				z=c;
		}
}

class FourD extends ThreeD{
		int t;
		FourD(int a, int b, int c, int d){
				super(a,b,c);
				t=d;
		}
}
```

```
class Coords <T extends TwoD>{
                    T [ ] coords;
                    Coords(T [ ] o){
                              coords=o;
                    }
          }

public class BoundedWildcard {
          static void showXY(Coords<?> c){
          System.out.println("X Y Coordinates: ");
          for(int i=0;i<c.coords.length;i++)
          System.out.println(c.coords[i].x + " " +c.coords[i].y);
          System.out.println();
          }

          static void showXYZ(Coords<? extends ThreeD> c){
                    System.out.println("X Y Z Coordinates: ");
                    for(int i=0;i<c.coords.length;i++)
                    System.out.println(c.coords[i].x + " " +c.coords[i].y+ " " +c.coords[i].z);
                    System.out.println();
                    }

          static void showAll(Coords<? extends FourD> c){
                    System.out.println("X Y Z T Coordinates: ");
                    for(int i=0;i<c.coords.length;i++)
System.out.println(c.coords[i].x + " " +c.coords[i].y+ " " +c.coords[i].z+ " " +c.coords[i].t);
                    System.out.println();
                    }
public static void main(String[ ] args) {
            TwoD [ ] td ={new TwoD(0,0), new TwoD(7,9), new TwoD(18,4),
new TwoD(-1,-2)};

            Coords <TwoD> td1ocs = new Coords< > (td);
                        System.out.println("Contents of td1ocs");
                        showXY(td1ocs);
                        //          showXYZ(td1ocs);err not ThreeD
                        //          showAll(td1ocs);err not FourD
            FourD [ ] fd= {new FourD(1,2,3,4),
                                    new FourD(6,8,14,8),
                                    new FourD(22, 9, 7, 5),
                                    new FourD(-3, 5, 7, -17)
                        };
            Coords <FourD> fd1ocs = new Coords< > (fd);
                        System.out.println("Contents of fd1ocs");
                        showXY(fd1ocs);
                        showXYZ(fd1ocs);
                        showAll(fd1ocs);
                        }//main
}//BoundedWildcard
```

Generic method in non Generic class

```
//G6
public class GenMethNonGenClass {

        static <T, V extends T> boolean isIn (T x, V[ ] y){
                for (int i=0;i<y.length;i++)
                        if(x.equals(y[i]))return true;
                        return false;}

        public static void main(String[ ] args) {
        Integer nums[ ] ={1, 2, 3, 4, 5};
        if (isIn(2, nums))System.out.println("2 is in nums");
        if (!isIn(7, nums))System.out.println("7 is not in nums");
        System.out.println();
        String [ ]strs = {"one", "two", "three", "four", "five"};
        if (isIn("two", strs))System.out.println("two is in strs");
if (!isIn("seven", strs))System.out.println("seven is not in strs");
        }//main
}//GenMethNonGenClass
```

Generic constructor

```
//G7
class GenCons{
        private double val;
        <T extends Number> GenCons(T arg){
                val=arg.doubleValue( );
        }
        void showAll( ){
                System.out.println("val= "+ val);
        }
}//GenCons

public class GenConsDemo {
        public static void main(String[ ] args) {
                GenCons test = new GenCons(200);
                GenCons test2 = new GenCons(177.17F);
                test.showAll();
                test2.showAll();
        }//main
}//GenConsDemo
```

Generic interfaces

```
//G8
interface MinMax <T extends Comparable<T>>{
        T min();
        T max();
}//MinMax

class MyClass <T extends Comparable <T>> implements MinMax<T>{
        T[ ] vals;
        MyClass (T [ ] o){vals=o;
                        }
        public T min( ){T v = vals[0];
                for(int i=1;i<vals.length;i++)
                        if(vals[i].compareTo(v)<0)v=vals[i];
                return v;
                }
        public T max( ){T v = vals[0];
                for(int i=1;i<vals.length;i++)
                        if(vals[i].compareTo(v)>0)v=vals[i];
                return v;
                }
}//MyClass
```

```
public class GenIFDemo {
        public static void main(String[ ] args) {
Integer inums[ ] = {3,6,2,7,8};
Character chs[ ] = {'a', 'b', 'c', 'd'};

MyClass <Integer> iob = new MyClass< > (inums);
MyClass <Character> cob = new MyClass< > (chs);

System.out.println("Max value in inums = "+ iob.max( ));
System.out.println("Min value in inums = "+ iob.min( ));

System.out.println("Max value in chs = "+ cob.max( ));
System.out.println("Min value in chs = "+ cob.min( ));
        }//main
}//GenIFDemo
```

Raw types and generics

- There is a problem with a ***transition to*** the ***non-generic*** Java code *and* ***vice versa***, the generic to support the old code.
- This allows a generic class to be used without any specified type argument, which creates a so-called *raw-type* for the class. This *raw-type* is compatible with *legacy-code* methods that do not know anything generic. The disadvantage is that the ***typesafety*** **of generics is lost**.

- In the next example *raw* is actually an *Object* object, it is not *typesafe* and can be assigned to any *raw* reference type of a *Gen* object and vice versa. This results in execution, *run-time errors*, not compilation errors.
- Because of potential threats inherent to *raw* types, the Java compiler will display:

"Unchecked warnings" when such situations occur.

```
//G9
public class RawDemo {
        @SuppressWarnings({ "rawtypes", "unchecked" })
        public static void main(String[ ] args) {
        Gen <Integer> iOb = new Gen <> (88);
        Gen <String> strOb = new Gen<> ("Generic String Test");
        Gen  raw = new Gen <Double> (98.7);// raw of Double
        double d = (Double)raw.getOb();//Cast type unknown
        System.out.println("value of d = " +d);
        int i = iOb.getOb();
        System.out.println("value of i = " +i);
        String s = strOb.getOb();
        System.out.println("value of s = " +s);

        //int id= (Integer) raw.getOb();//run-time error, raw is Double
        //System.out.println("value of id = " +id);
        strOb=raw;//Ok, potentially wrong
        //String sraw = strOb.getOb();
        //System.out.println("value of sraw = " +sraw);
        raw=iOb;//Ok, potentially wrong
                        }//main
}//RawDemo
```

Generics inheritance

```
//G10
class Gen <T> {
        T ob;
        //cons
        Gen (T o){
                ob=o;
        }
        //get
        T getOb(){
                return ob;
        }
        //show
        void ShowType(){
        System.out.println("Type of T is "+ ob.getClass().getName());
        }
}//Gen
```

```
class Gen2 <T, V> extends Gen <T>
{
V ob2;
Gen2 (T o, V o2){super(o); ob2=o2;}
V getOb2 (){return ob2;}
}//Gen2

public class GenHierDemo{
        public static void main(String args[ ]){
Gen2 <String, Integer> X = new Gen2< > ("Value is = ", 99);
                System.out.print(X.getOb());
                System.out.println(X.getOb2());
        }//main
}//HD
```

Generics inheritance of non generic class

```
//G10.1
class NonGen {
        int num;
NonGen(int i){num=i;}
int getNum(){return num;}
}//NonGen

class GenG <T> extends NonGen {
        T ob;
        GenG(T o, int i){super (i); ob=o;}
        T getOb(){return ob;}
}//Gen

public class GenHierDemo2{
        public static void main(String args[ ]){
                GenG <String> W = new GenG<String> ("Hello= ", 100);
                System.out.print(W.getOb());
                System.out.println(W.getNum());
        }//main
}//HD2
```

Collections with Generics

So far we have only been discussing collections *without* generic types.

-Collections *without* generics (in pre-JDK 1.2) may be easier to use if you already know the type that will be dealt with inside the collections interface.

-Collection implementations in pre-JDK 1.2 versions of the Java platform included few data structure classes but did not contain a collection framework.

Collections *with* generics are used when you are *uncertain what type* is inside the interface or if you wish for the interface to be *compatible with many types*.

Starting with JDK 5.0 we discuss about a **Java Collection Framework** that provides an architecture to store and manipulate a group of objects.

A Java collection framework includes the following:

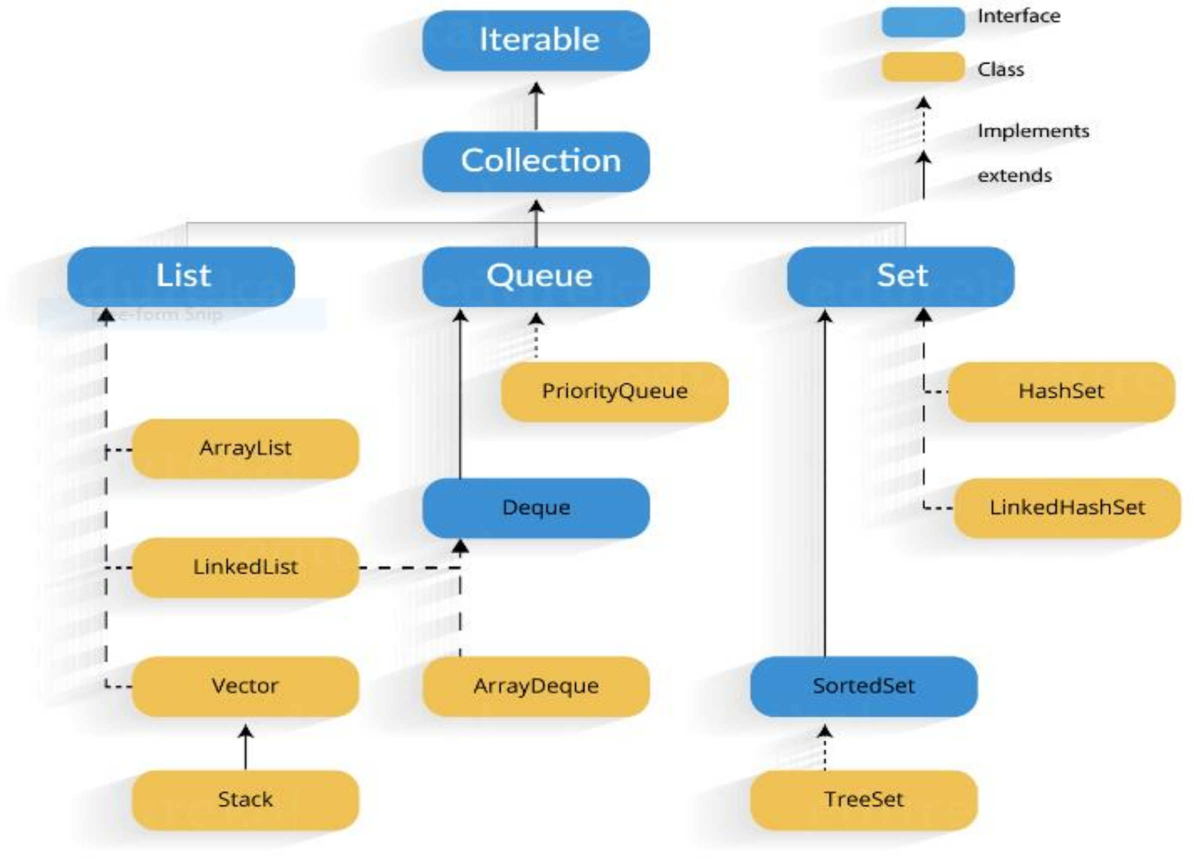
-Interfaces

-Classes

-Algorithms,

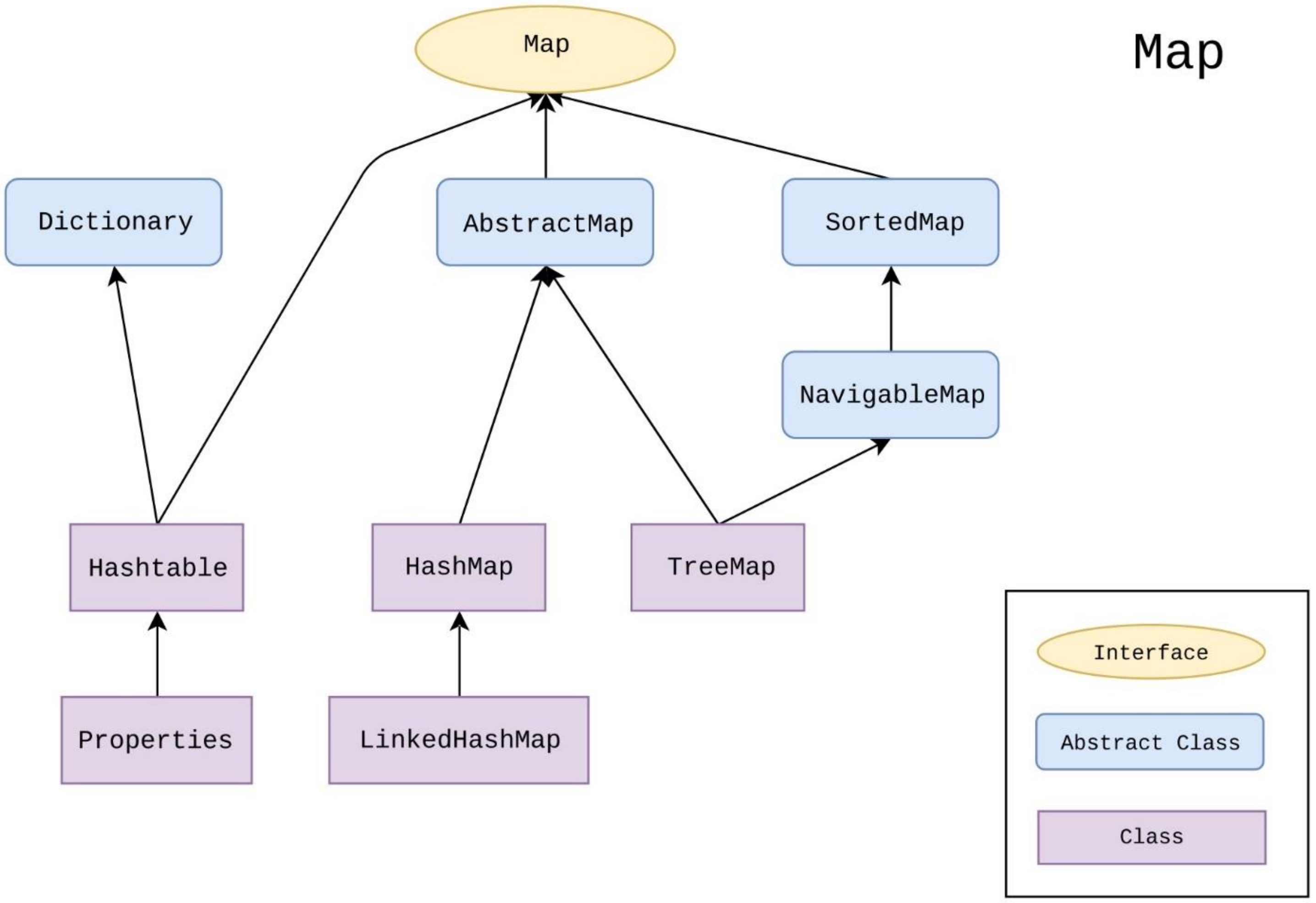
localized in *java.util* package.

Interfaces and Classes in Java Collection Framework



Map Collection

Map



Methods of the Collection Interface

- The *Collection* interface is the interface which is implemented by all the classes in the collection framework.
- It declares the methods that every collection will have.
- In other words, we can say that the *Collection* interface builds the foundation on which the collection framework depends.
- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html

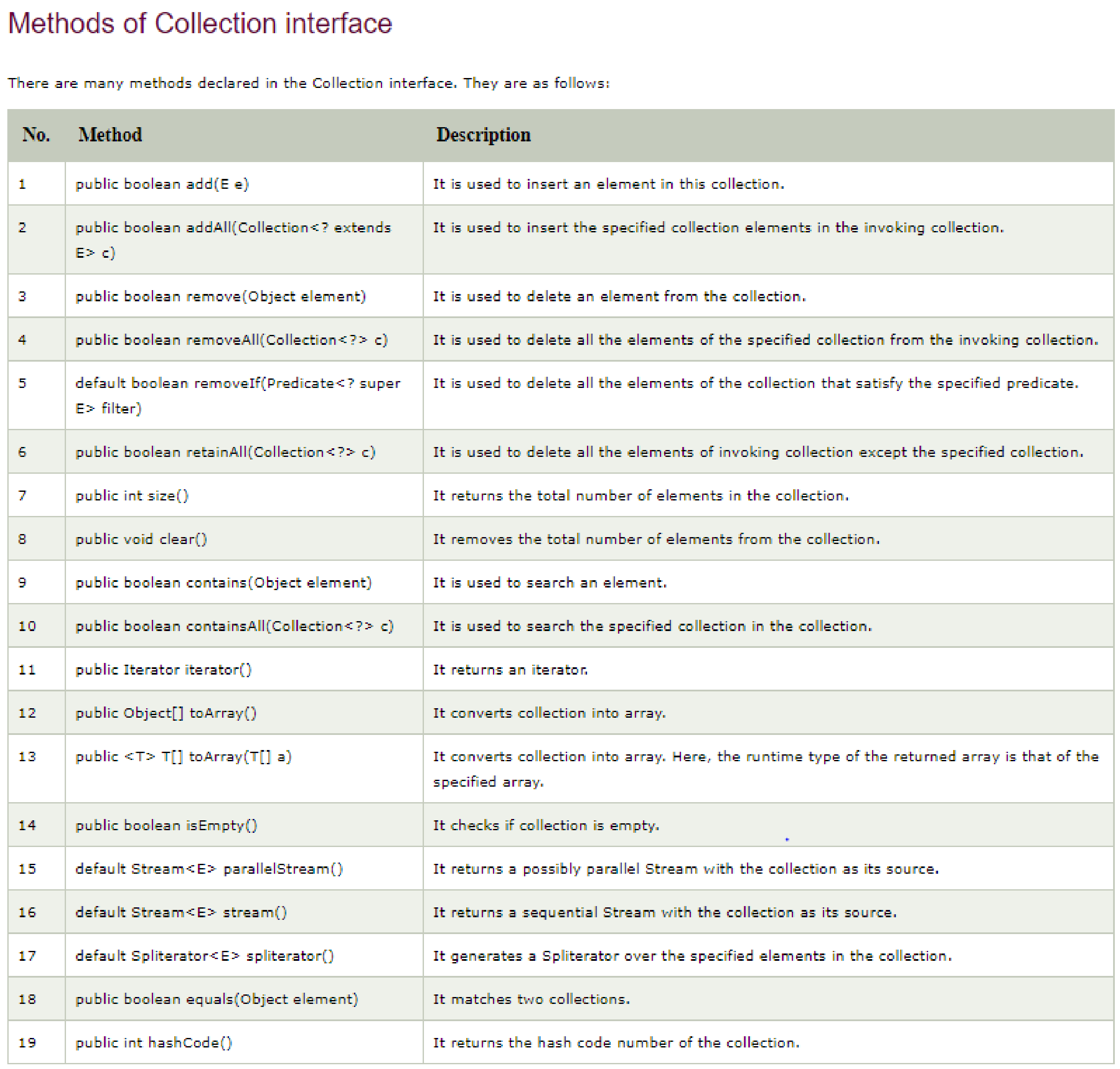
- Some of the methods of *Collection* interface are:

-boolean add (E e),

-boolean addAll (Collection <? extends E> c),

-void clear(), etc.

which are implemented by all the subclasses of *Collection* interface.



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterable Interface

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

public interface Collection<E>extends Iterable<E>

- The *Collection* interface contains only one abstract method. i.e. (that must be defined implicit/explicit by classes that implements the interface),

    Iterator<T> iterator() ;

It returns the iterator over the elements of type *T*.

-*forEach ()* method was introduced in *new Java LTS versions* to iterate on a collection like *for_loop* instruction and belongs to *Iterable* interface.

Iterator interface

- *Iterator* takes the place of *Enumeration* in the *Java Collections Framework*.
- There are only three methods in the *Iterator* interface.

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

They are used to iterate through a collection.

Scanner class implements *Iterator* interface.

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

Ordering a Collection

We may use ***Comparable*** interface:

```
interface Comparable<T> {
      public int compareTo(T o);
 }
```

It is also possible to use ***Comparator*** interface with the methods:

```
naturalOrder( ), reverseOrder( ), comparing( ),
compare( ), …
```

-The ***Comparable*** interface is a good choice when used for defining the **default ordering** or, in other words, if it's the main way of comparing objects.

-Using ***Comparator*** interface it allows us to avoid **adding additional code** to our domain classes.

We can define multiple different comparison strategies which isn't possible when using *Comparable.*

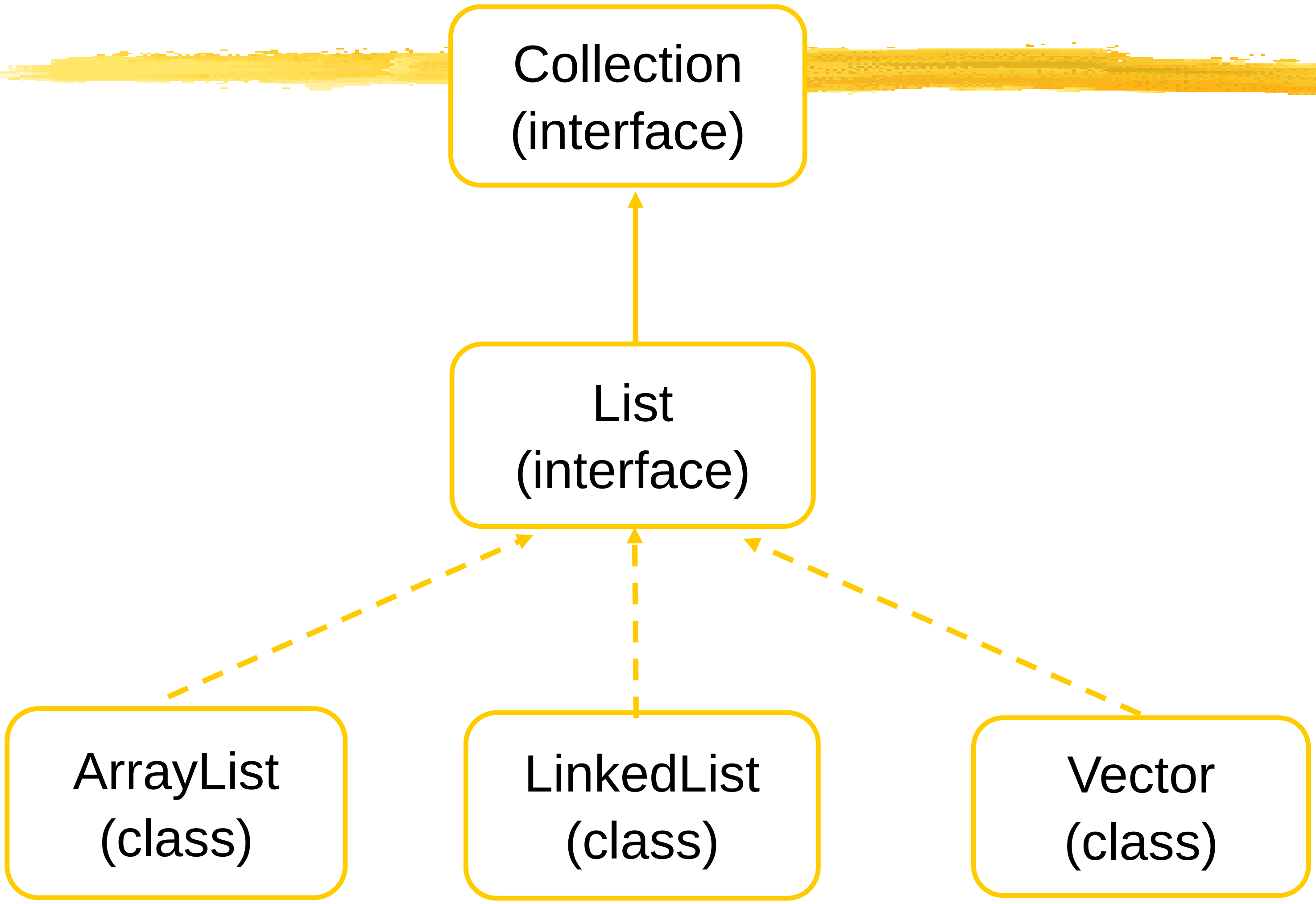
Lists

A list, in Java, is an ordered collection that may contain duplicate elements. In other words, *List* extends *Collections*.

Important qualities of *Lists* are:

- They grow and shrink as you add and remove objects.
- They maintain a specific order.
- They allow duplicate elements

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html


Collection
(interface)
List
(interface)
ArrayList
(class)
LinkedList
(class)
Vector
(class)

- **Problem**

– You want to make an ordered list of objects. But, even after you get the first few elements, you don’t know how many more you will have.

- Thus, you can’t use an array, since the size of arrays must be known at the time that you allocate it. (Although Java arrays are better than C++ arrays since the size does not need to be a compile-time constant)
- **Solution**

– Use *ArrayList* or *LinkedList*: they stretch as you add elements to them

- **Notes**

– The two options give the same results for the same operations, but differ in performance

ArrayList's

Until now we have been using arrays inside a collections class to create our collections interface.

When you use arrays, you are restricted to the size of the array that you initiate inside the constructor.

ArrayList's are very similar to arrays except that you do not need to know the size of the *ArrayList* when you initialize it like you would an array. You may alter the size of the *ArrayList* by adding or removing elements. The only information you need when initializing an *ArrayList* is the **object type** that it stores.

The following code initializes an *ArrayList* of Cards.

ArrayList<Card> Deck = new ArrayList<Card>( );//before Java 7
or ,
ArrayList<Card> Deck = new ArrayList< >( );//>=Java 7

- The *ArrayList* class already exists in Java, and it contains many methods including the following:

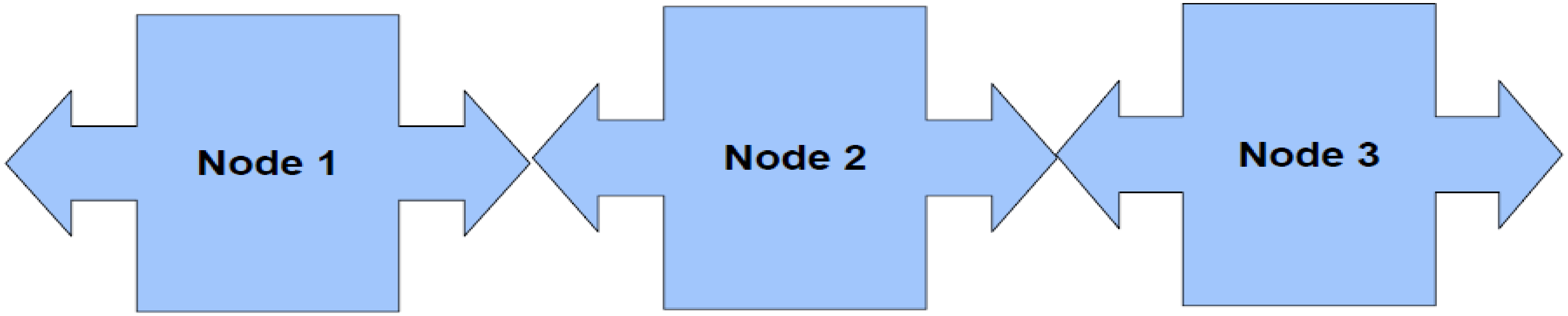
Method	Method Description
`add(ObjectType t)`	Adds an element of type *ObjectType* to the end of the arraylist.
`add(ObjectType t, int i)`	Adds an element of type *ObjectType* to index *i* of the arraylist.
`get(int i)`	Returns the element of the arraylist at index *i*.
`set(int i, ObjectType t)`	Assigns the element of index *i* with the value of element *t*.
`remove(int i)`	Removes the element at index *i* from the arraylist and returns that element.
`remove(ObjectType t)`	Searches through the arraylist until the element *t* is found, removes it and returns true if the element is found, returns false if the element is not found.

LinkedLists

LinkedList as a brief presentation is a list of elements that is dynamically stored. (A *Double LinkedList – DLL* will be used)

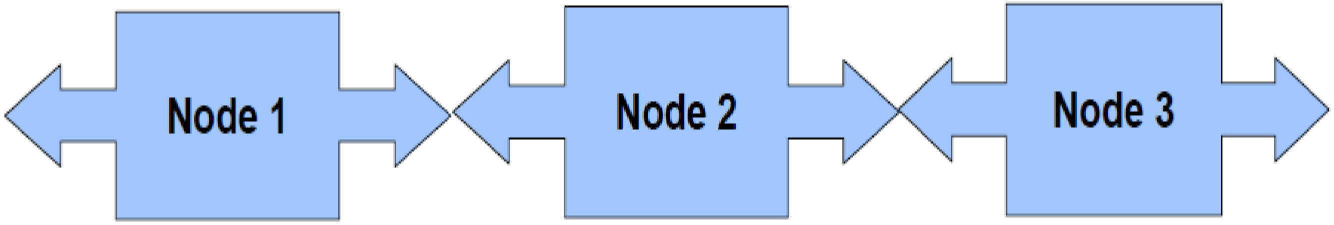
Like an *ArrayList*, it changes size and has an explicit ordering. But it **doesn't use an array to store information.**

It uses an object known as a Node. Nodes are like roadmaps: they tell you where you are (the element you are looking at), and where you can go (the previous element and the next element).

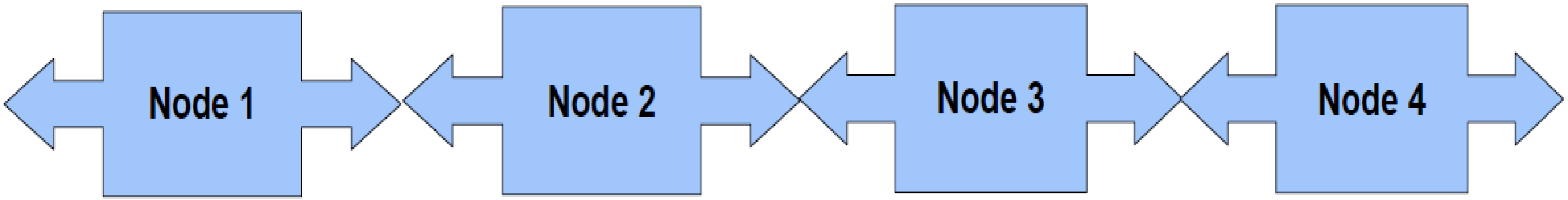


-Ultimately, we have a list of *Nodes*, which point to other *Nodes* and have an element attached to them. When we want to add a *Node*, we simply set its left *Node* to the one on its left, and its right *Node* to the one on its right. But don't forget to change the *Nodes* around it too!

Let's say we wanted to add a 4th node to the end of this linked list.



It would look like this.



A *LinkedList* is initialized in the same way as *ArrayList*'s.

The following code shows how to initialize a linked list of pancakes.

LinkedList<Pancake> myStack = new LinkedList<>( );

FIFO LinkedList Methods	LIFO LinkedList Methods
`add(Object o)` //appends the given element to the end of the list	`add(Object o)` //appends the given element to the end of the list
`removeFirst()` //removes the first element from the list and returns it	`removeLast()` //removes the last element from the list and returns it
`getFirst()` //returns the first element of the list	`getLast()` //returns the last element of the list

Syntax: ArrayList & LinkedList

- **Summary of operations**

– Create empty list

- **new ArrayList< >( )** or **new LinkedList< >( )**

– Note that you need "*import java.util.*;*" at the top of file

– Add entry to end

- **add(value)** (adds to end) or **add(index, value)**

– Retrieve *n*-th element

- **get(index)**

– Check if element exists in list

- **contains(element)**

– Remove element

- **remove(index)** or **remove(element)**

– Find the number of elements

- **size()**

ArrayList Example

```
import java.util.*; // Don't forget this import
public class ListTestAL {
public static void main(String[ ] args) {
List<String> entries = new ArrayList<>( );
double d;
while((d = Math.random()) > 0.1) {
        entries.add("Value: " + d);
                }
for(String entry: entries) {
System.out.println(entry);}
        }//main
}//class
```

List<String> tells Java your list will contain only strings.

Before Java 7, you had to repeat the type here, using ***new ArrayList<String>( )*** instead of ***new ArrayList<>( )***

: Output

```
> java ListTest2
Value:
0.6374760850618444
Value:
0.9159907384916878
Value:
0.8093728146584014
Value:
0.7177611068808302
Value:
0.9751541794430284
Value:
0.2655587762679209
Value:
0.3135791999033012
Value:
0.4462415271013836
Value:
0.7585420756498766
```

Accessing with an iterator:

```
import java.util.*;
public class ListTestAL_Iter {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<>();
    double d;
    while((d = Math.random()) > 0.1)
                    entries.add("Value_Iterator: " + d);

    Iterator<String> listIterator = entries.iterator();
    while (listIterator.hasNext()) {
                    System.out.println(listIterator.next());
    }
  }//main
}//class
```

Accessing with *forEach( )* method:

```
import java.util.*;

public class ListTestAL_forEach {
  public static void main(String[ ] args) {
        List<String> entries = new ArrayList<>( );
        double d;
        while((d = Math.random()) > 0.1)
              entries.add("Value_forEach: " + d);

        entries.forEach((temp) -> {
              System.out.println(temp);
              });

        }//main
}//class
```

```
void forEach(Consumer<? super T> action)
@FunctionalInterface
public interface Consumer {
   void accept(T t);  }
```

More List Methods: Code

```
import java.util.*; // Don't forget this import
public class MoreWithLists {
public static void main(String[ ] args) {
List<String> names = new ArrayList<>( );
// or List<String> names = new LinkedList<>( );
names.add("Marty");
names.add("Cay");
names.add("David");
int numEntries = names.size( );
System.out.println("Num entries: " + numEntries);
String secondEntry = names.get(1);
System.out.println("2nd entry: " + secondEntry);
boolean containsJoe = names.contains("Joe");
System.out.println("Contains Joe? " + containsJoe);
names.remove("Marty");
numEntries = names.size( );
System.out.println("Num entries: " + numEntries);
            }//main
}//class
```

Output:

- **Output when using ArrayList**

```
Num entries: 3
2nd entry: Cay
Contains Joe? false
Num entries: 2
```

- **Output when using LinkedList (same!)**

```
Num entries: 3
2nd entry: Cay
Contains Joe? false
Num entries: 2
```

ArrayList Example: Explicit Typecasts (Java 1.4 and Earlier) – no generics

```
import java.util.*;
public class ListTest1 {
@SuppressWarnings("unchecked")
public static void main(String[ ] args) {
@SuppressWarnings("rawtypes")
List entries = new ArrayList( );
double d;
while((d = Math.random( )) > 0.1)
        entries.add("Value:  " + d);
String entry;
for(int i=0; i<entries.size( ); i++) {
entry = (String)entries.get(i);
System.out.println(entry);}
        }//main
 }
```

ArrayList Example: Generics (>Java 7)

```
import java.util.*;
public class ListTest2 {
public static void main(String[ ] args) {
List<String> entries = new ArrayList<>();
double d;
while((d = Math.random( )) > 0.1)
         entries.add("Value:  " + d);
for(String entry: entries) {
System.out.println(entry);}
         }//main
}
```

List<String> tells Java your list will contain only strings. Java will check at *compile* time that all additions to the list are of *String* type. You can then use the simpler looping construct because Java knows ahead of time that all entries are of *String* type.

ArrayList Example: (Java 5 and 6)

```
import java.util.*;
public class ListTest3 {
public static void main(String[ ] args) {
List<String> entries = new ArrayList<String>();
double d;
while((d = Math.random( )) > 0.1)
        entries.add("Value:  " + d);
for(String entry: entries) {
System.out.println(entry);}
        }//main
}
```

Before Java 7, you had to use ***ArrayList<String>*** instead of ***ArrayList<>*** here. In Java 7 and later, the compiler will do type inferencing. But you still need ***List<String>*** in the variable declaration.

Sorting a *List* with *Comparable* Interface

```
import java.util.*;

class NameS implements Comparable<NameS> {
private final String firstNameS, lastNameS;
public NameS(String firstNameS, String lastNameS){
this.firstNameS = firstNameS;
this.lastNameS = lastNameS;}
public String getFirstNameS(){ return firstNameS;}
public String getLastNameS(){ return lastNameS;}
@Override
public boolean equals(Object o) {//override din clasa Object pentru clasa NameS
if (!(o instanceof NameS))  return false;
NameS temp = (NameS) o;
return temp.firstNameS.equals(firstNameS) &&
temp.lastNameS.equals(lastNameS);}
@Override
public int hashCode() {//override din clasa Object pentru clasa NameS
     return firstNameS.hashCode() + lastNameS.hashCode();}
@Override
 public String toString() {  //override din clasa Object pentru clasa NameS
           return firstNameS + " " + lastNameS;}
//metoda redefinita din interfata Comparable pt. lastNameS & firstNameS
@Override
public int compareTo(NameS n) {
int lastCmp = lastNameS.compareTo(n.getLastNameS());
return (lastCmp != 0 ? lastCmp : firstNameS.compareTo(n.getFirstNameS()));
 }
}// class NameS

public class SortListComparable {
 public static void main(String... args) {
            NameS bandsArray[ ] = {
                                    new NameS("Foo", "Fighters"),
                                    new NameS("Nir", "Vana"),
                                    new NameS("Met", "Allica"),
                                    new NameS("AC", "DC")    };
            List<NameS> NameSs = Arrays.asList(bandsArray);
            Arrays.sort(bandsArray);//ordonare compareTo()
            //Collections.sort(NameSs);//ordonare compareTo()
            System.out.println(NameSs);
NameS ob= new NameS("Nir", "Vana");
System.out.println("NameSs contains Nirvana = "+NameSs.contains(ob)); //needs
equals() & hashCode()
            }// main
} //SortListComparable
```

```
//sortare campuri de tipuri diferite, cu compareTo( ) si compare( ), comparing,
//stream.sorted(), …
import java.util.*;
import java.util.stream.Collectors;
import java.lang.Comparable;

class Name implements Comparable<Name> {
                    String firstName, lastName;
                    int id;
                    double weight;
          public Name(String firstName, String lastName, int id, double weight) {
                    this.firstName = firstName;
                    this.lastName = lastName;
                    this.id=id;
                    this.weight=weight;
          }
          public String getFirstName() { return firstName; }
          public String getLastName()  { return lastName;  }
          public int getId()  { return id;  }
          public double getWeight()  { return weight;  }

          @Override
          public String toString() {
                    return firstName + " " + lastName+ " "+ id+ " "+ weight;
          }
          @Override
          //lastName, String
          public int compareTo(Name n) {
                    int lastCmp = lastName.compareTo(n.getLastName());
                    return lastCmp;
          }

          //static objects and methods
          //firstName, String
          public static Comparator<Name> FirstNameComparator = new
Comparator<>() {
                    public int compare(Name name1, Name name2) {
                              int firstCmp =
name1.getFirstName().compareTo(name2.getFirstName());
                              return firstCmp;
                    }};

                    //id, int
          public static Comparator<Name> IdComparator = new Comparator<>() {
                              public int compare(Name name1, Name name2) {
                                        int idCmp = name1.getId()-name2.getId();
                                        return idCmp;
                              } };
```

```
        //weight, double
        public static Comparator<Name> WeightComparator = new
Comparator<>() {
                public int compare(Name name1, Name name2) {
                        int weightCmp;
        if(name1.getWeight()==name2.getWeight()) weightCmp=0;
        else if(name1.getWeight()>name2.getWeight()) weightCmp=1;
                        else  weightCmp=-1;
                return weightCmp;
                } };

                //firstName & id, String&int
                public static int FirstNameThenId(Name lhs, Name rhs) {
        if (lhs.firstName.equals(rhs.firstName)) return lhs.id - rhs.id;
                   else return lhs.firstName.compareTo(rhs.firstName);
                }
        }//Name
```

```
public class ListSortMultiple            {
public static void main(String... args) {
        Name bandsArray[] = {
                new Name("Foo", "Fighters", 1, 1.1),
                new Name("Nir", "Vana", 2, 7.7),
                new Name("Met", "Allica", 3, 29.3),
                new Name("Met", "Allica", 3, 3.3),
                new Name("AC", "DC", 5, 9.9),
                new Name("AC", "DC", 4, 19.9)
        };
        //equivalent List <Name>
List<Name> names = Arrays.asList(bandsArray);
System.out.println("Initial List from bandsArray: "+names);
        //implicit compareTo() by lastName: String
        Arrays.sort(bandsArray);
        List<Name> sortedLastNames = Arrays.asList(bandsArray);
        System.out.println("Sorted after last name: "+sortedLastNames);

        //lambdas lastName: String
        List<Name> LsortedLastNames = Arrays.asList(bandsArray);
        LsortedLastNames.sort((h1, h2) ->
h1.getLastName().compareTo(h2.getLastName()));
        System.out.println("Sorted after last name lambda expression:
"+LsortedLastNames);
```

```
        //static Comparator - compare() firstName: String
                Arrays.sort(bandsArray, Name.FirstNameComparator);
                List<Name> sortedFirstNames = Arrays.asList(bandsArray);
                System.out.println("Sorted after first name: "+sortedFirstNames);

                //static Comparator - compare() id: int
                Arrays.sort(bandsArray, Name.IdComparator);
                List<Name> sortedId = Arrays.asList(bandsArray);
                System.out.println("Sorted after Id: "+sortedId);

                //static Comparator - compare() weight: double
                Arrays.sort(bandsArray, Name.WeightComparator);
                List<Name> sortedWeight = Arrays.asList(bandsArray);
                System.out.println("Sorted after Weight: "+sortedWeight);

                //reference to static method - compareTo() firstName & id: String&int
                List<Name> namesFId = Arrays.asList(bandsArray);
                namesFId.sort(Name::FirstNameThenId);
        System.out.println("Sorted after FirstNameThenId: (Method Reference)"+namesFId);
                //Sorted Reverse after LastName lambda expression: String:
                 Comparator<Name> comparatorR = (h1, h2) ->
h1.getLastName().compareTo(h2.getLastName());
            LsortedLastNames.sort(comparatorR.reversed());
            System.out.println("Sorted Reverse after LastName lambda expression:
"+LsortedLastNames);
```

```
            //Comparator Collections comparing() weight: double
                        Collections.sort(names,
Comparator.comparingDouble(Name::getWeight));
            System.out.println("Sorted after Weight comparing (Method Reference): "+names);

            //Comparator Multiple Conditions: firstName, Id, weight: String&int&double
                        List<Name> namesFIdWeight = Arrays.asList(bandsArray);

namesFIdWeight.sort(Comparator.comparing(Name::getFirstName).thenComparing(Name::get
Id).thenComparing(Name::getWeight));
            System.out.println("Sorted after FirstNameThenIdThenWeight (Method Reference):
"+namesFIdWeight);

            //Sorting a List in Reverse with stream.sorted() and lambdas, lastName: String
                        Comparator<Name> reverseNameComparator = (h1, h2) ->
h2.getLastName().compareTo(h1.getLastName());
                        List<Name> reverseSortedNames =
names.stream().sorted(reverseNameComparator).collect(Collectors.toList());
                        System.out.println("Sorted reverse after lastName with stream.sorted() :
"+reverseSortedNames);

}
            }
```

Sets

-A *Set* is a collection of elements that does not contain duplicates. For example, a set of integers 1, 2, 2, 3, 5, 3, 7 would be:

{1, 2, 3, 5, 7}

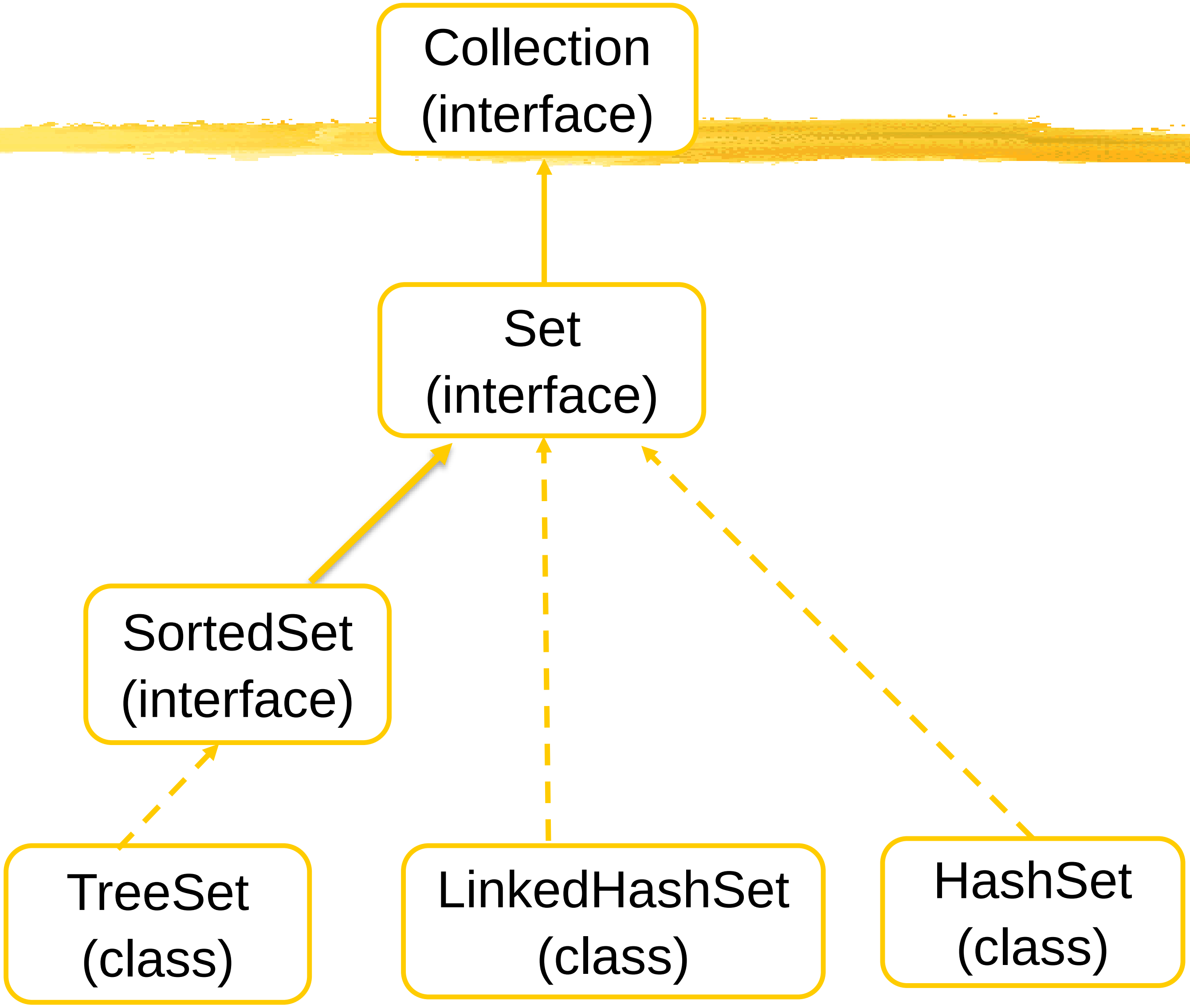
-All elements of a set must be of the same type.

For example, you can not have a set that includes integers and strings. But you could have a set of integers and a separate set of strings.

-We've already seen how *Lists*, which are a collection that may contain duplicates, are implemented through *ArrayLists*.

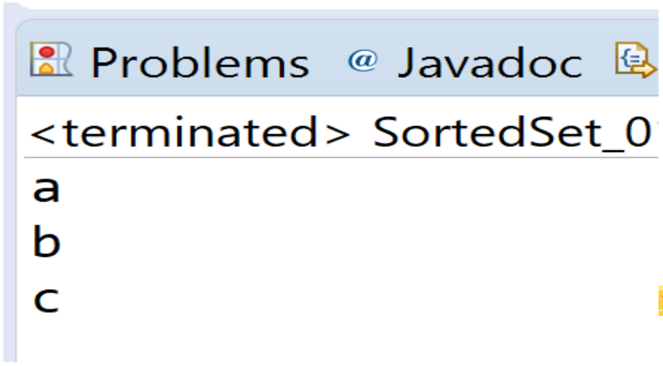
-Similarly, *Sets* are commonly implemented with *TreeSet*, *LinkedHashSet*, or *HashSet*.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Set.html


Collection
(interface)
Set
(interface)
SortedSet
(interface)
TreeSet
(class)
LinkedHashSet
(class)
HashSet
(class)

Example TreeSet

```
import java.util.*;
public class SortedSet_01 {
public static void main(String[ ] args) {
        // crearea unui set sortat TreeSet
        SortedSet <String>set = new TreeSet<>( );
        // adaugarea de elemente in set
        set.add("b");
        set.add("c");
        set.add("a");
        // parcurgerea elementelor din set
        Iterator <String>it = set.iterator( );
        while (it.hasNext( )) {
                // preluarea elementului curent
                String element = it.next( );
                System.out.println(element);}
                }//main
        } //class
```



```
<terminated> SortedSet_0
a
b
c
```

HashSets

A *HashSet* is similar to an *ArrayList* but **does not have any specific ordering.**

Example

-Imagine you have 35 coins in a bag. **There is no special ordering of these coins in this bag**, but we can search the bag at any time to see if it contains a certain coin. We can add coins to the bag, we can remove coins from the bag, and we always know how many coins are in the bag.

-Now, think of the bag as the *HashSet.* There is no ordering and therefore no indexes of the coins, so we cannot increment through or sort *HashSets* because even if we did, with one little shake of the bag the order is lost.

HashSets have no guarantee that the order will be the same throughout time.

HashSets are good to use when **order is not important**.

The code below demonstrates the initialization of *HashSet bagOfCoins*

HashSet<Coin> bagOfCoins = new HashSet<>( );

-The *HashSet bagOfCoins* is a set of objects of *Coin* type. This assumes that the class *Coin* has already been created.

-Although *HashSets* do not have ordering, we can search through them (**direct search**), just like we could search through the bag of coins to see if the coin we are looking for is in the bag.

-To search for a coin in the *HashSet bagOfCoins*, you would

use *HashSet*'s *contains(element)* overriding the *equals()* and *hashCode()* methods for the class where *element* object belongs:

bagOfCoins.contains(quarter);

//returns true if bagOfCoins contains the coin

More HashSet Methods

add(Object x);

Adds the object *x* to the *HashSet* of the same object type if it is not already present in the set.

remove(Object x);

Removes the object from the *HashSet*.

size( );

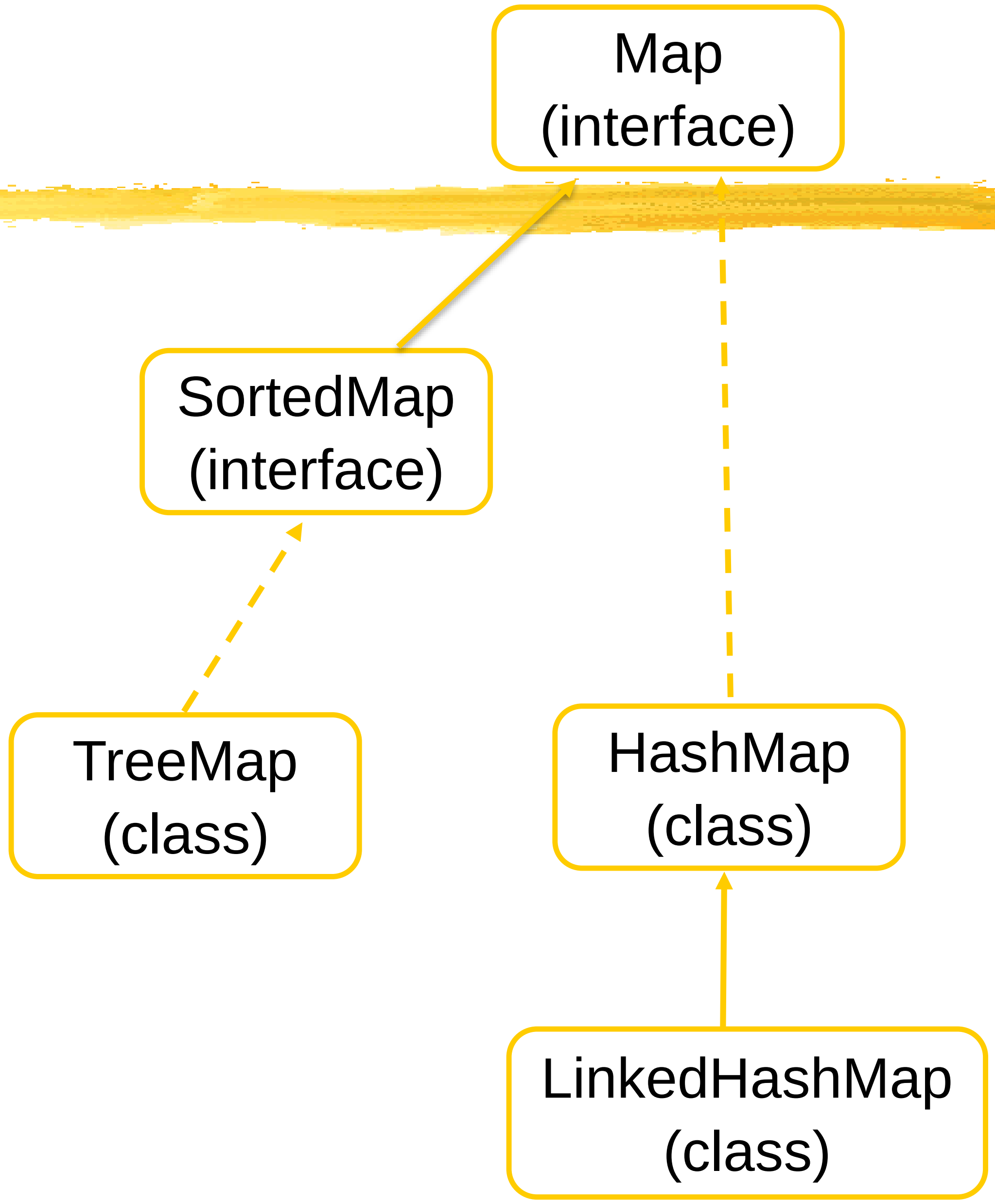
Returns the number of elements in the set.

Maps - (Also called “Lookup Tables” or “Associative Arrays”)

A *Map* is a collection that links a key to a value. Similar to how an array links an index to a value, a map links a key (one object) to a value (another object).

Maps, like sets, cannot contain duplicates. This means each key can only exist once and can only link to a single value.

Since *Map* is an interface, you must use one of the classes that implement *Map* such as *HashMap* to instantiate a *Map*.


Map
(interface)
SortedMap
(interface)
TreeMap
(class)
HashMap
(class)
LinkedHashMap
(class)

HashMaps

HashMaps are *Maps* that **link a *Key* to a *Value***.

The *Key* and *Value* can be of any type, but their types must be consistent for every element in the *HashMap*. Below is a generic breakdown of how to initialize a *HashMap:*

HashMap<KeyType,ValueType> mapName = new HashMap<>( );//after Java7

-*HashMap* - like a *Hashtable*, differences: unsynchronized, accept *null* elements (keys and values)

-*Hashtable* - keys must implement the *hashCode( )* method and the *equals( )* method, does not support *null*

-TreeMap - ordered by key, does not accept *null* keys, but accepts *null* values

-Let's say we wish to group together many different fruits

and wish to be able to store and later retrieve their color.

The first step to do this is initializing a *HashMap*.

HashMap<String,String> fruitBowl = new HashMap<>( );

To add fruits to our *fruitBowl*, simply use the *put(Key,Value)* method of *HashMaps*. Let's add a few fruits to our *fruitBowl:*

fruitBowl.put("Apple", "Red");

//adds the key *Apple* and its value *Red* to the *HashMap*

fruitBowl.put("Orange", "Orange");

//adds the key *Orange* and its value *Orange* to the *HashMap*

fruitBowl.put("Banana", "Yellow");

//adds the key *Banana* and its value *color Yellow* to the *HashMap*

The *Key* of a *HashMap* can be thought of as the *index* linked to the element (even though it does not necessarily have to be an integer)

Getting the value stored is easy once we understand that the *key* is the *index:* use the *get(Key)* method of *HashMap*.

Let's say we wanted to get the *color* of the Banana in the fruit bowl. We could do the following:

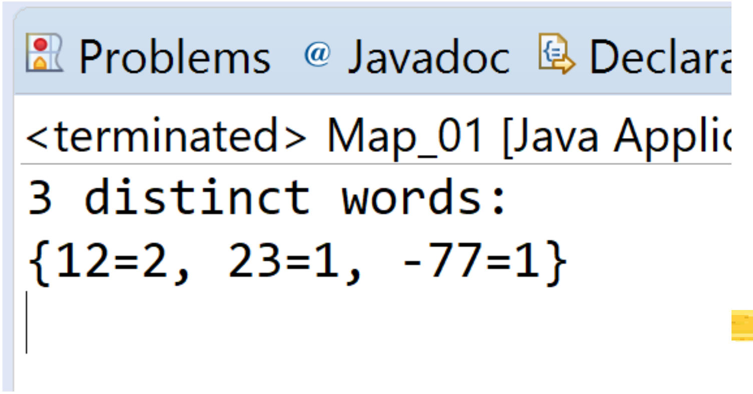
String bananaColor = fruitBowl.get("Banana");

This method searches through the *HashMap* until it finds a *Key* match to the parameter (until it finds "Banana") and returns the *Value* for that *Key* (returns "Yellow").

More HashMap Methods

HashMap Method	Description
`containsKey(KeyType Key)`	Returns true if the HashMap contains the specified Key.
`containsValue(ValueType Value)`	Returns true if the HashMap contains the specified value.
`keySet()`	Returns a set of the keys contained in the HashMap.
`values()`	Returns a collection of the values contained in the HashMap.
`remove(KeyType Key)`	Removes the specified key from the HashMap.
`size()`	Returns the number of key-value mappings in the HashMap.

Example HashMap


Problems @ Javadoc Declara

```
<terminated> Map_01 [Java Applic
3 distinct words:
{12=2, 23=1, -77=1}
```

```
import java.util.*;
public class HashMap_ex {
   public static void main(String[ ] args) {
      Map<String, Integer> map = new HashMap<>( );
      String values[] = {"12", "-77", "23", "12"};
      for (String a : values) {
            Integer freq = map.get(a);
            map.put(a,(freq==null)? 1: freq+1);
      }
      System.out.println(map.size() + " distinct words:");
      System.out.println(map);
   }
}
```

HashMap example with generics: Finding State Abbreviations Based on State Names

```
//MapTest
import java.util.*;

public class MapTest{ //MapTest
static Scanner inputScanner = new Scanner(System.in);
public static void main(String[ ] args){
StateMap states = new StateMap();
Map<String,String> stateAbbreviationTable =
states.getStateMap();
System.out.println("Enter state names. " + "Hit RETURN to quit");
String stateName;
String abbreviation;
while(true) {
System.out.print("State: ");
stateName = inputScanner.nextLine( );
if (stateName.equals("")) {
System.out.println("Come again soon.");
break;      }
abbreviation = stateAbbreviationTable.get(stateName);
if (abbreviation == null) {abbreviation = "Unknown";}
System.out.println(abbreviation);
                                }//while
                        }//main
                }//MapTest
```

```
class StateMap {
private Map<String,String> stateMap;
public StateMap() {
stateMap = new HashMap< >();
for(String[ ] state: stateArray) {
stateMap.put(state[0], state[1]);
                    }
          }
public Map<String,String> getStateMap( ) {
return(stateMap);
          }
public String[ ][ ] getStateArray( ) {
return(stateArray);
          }
private String[ ][ ] stateArray ={{"Alabama","AL"},
{"Alaska","AK"},{"Arizona","AZ"}};
}//StateMap
```

Queues

A *Queue* is a list of elements with a *First In First Out* (***FIFO***) ordering.

• When you en-queue an element, it adds it to the end of the list.

• When you de-queue an element, it returns the element at the front of the list and removes that element from the list.

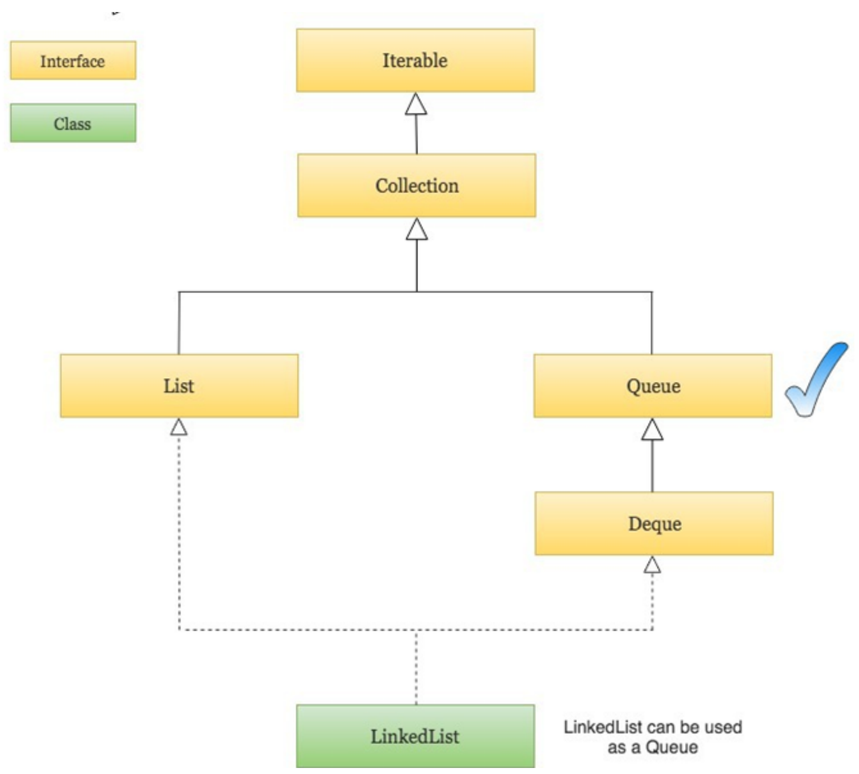
Java provides two new collection classes ***Queue*** and ***Deque*** to manage these dynamic collections.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Queue.html

Examples:

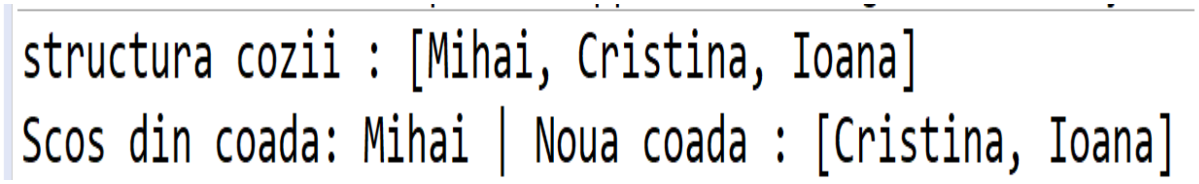
-Imagine a line at the movie theater, the first person there is the first person to get his/her ticket.

Ques classes and Interfaces



Queue Interface Example

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample {
  public static void main(String[] args) {
    Queue<String> waitingQueue = new LinkedList<>( );
    // adaugare elemente
    waitingQueue.add("Mihai");
    waitingQueue.add("Cristina");
    waitingQueue.add("Ioana");
    System.out.println("structura cozii : " +   waitingQueue);
    String name = waitingQueue.remove();
    System.out.println(" Scos din coada: " + name        + " |
Noua coada : " + waitingQueue);
}//main
}//class
```

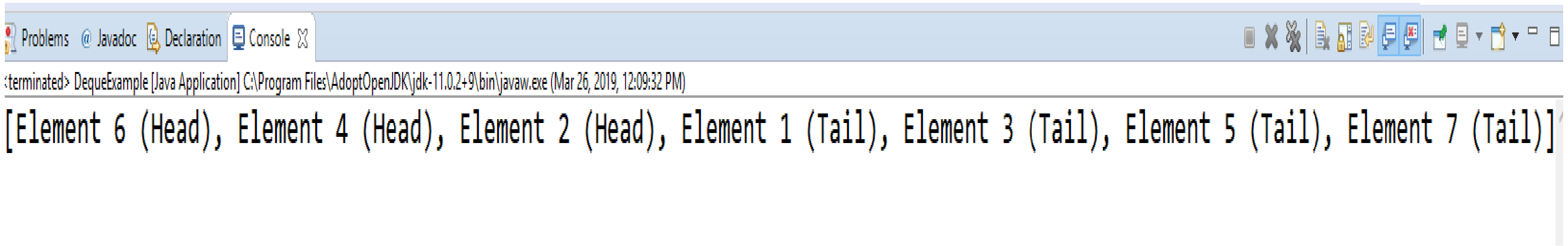


```
structura cozii : [Mihai, Cristina, Ioana]
Scos din coada: Mihai | Noua coada : [Cristina, Ioana]
```

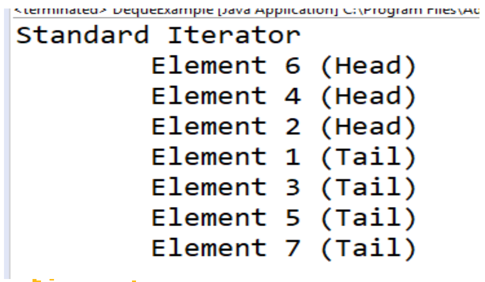
Deque Interface Example

```
import java.util.*;

public class DequeExample {
        public static void main(String[] args) {
        Deque<String> deque = new LinkedList<>( );
        deque.add("Element 1 (Tail)"); // adauga la baza
        deque.addFirst("Element 2 (Head)");
        deque.addLast("Element 3 (Tail)");
        deque.push("Element 4 (Head)"); //adauga la varf
        deque.offer("Element 5 (Tail)");
        deque.offerFirst("Element 6 (Head)");
        deque.offerLast("Element 7 (Tail)");
        System.out.println(deque + "\n");
```

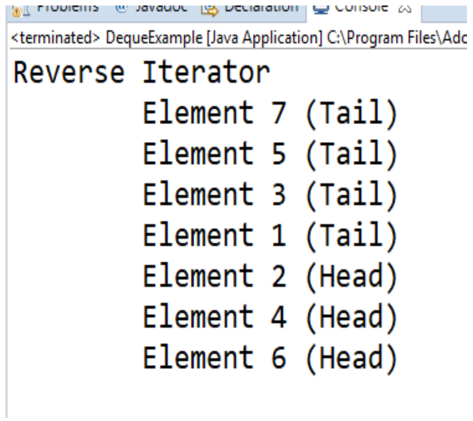


```
<terminated> DequeExample [Java Application] C:\Program Files\AdoptOpenJDK\jdk-11.0.2+9\bin\javaw.exe (Mar 26, 2019, 12:09:32 PM)
[Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
```



```
<terminated> DequeExample [Java Application] C:\Program Files\Ad
Standard Iterator
	Element 6 (Head)
	Element 4 (Head)
	Element 2 (Head)
	Element 1 (Tail)
	Element 3 (Tail)
	Element 5 (Tail)
	Element 7 (Tail)
```

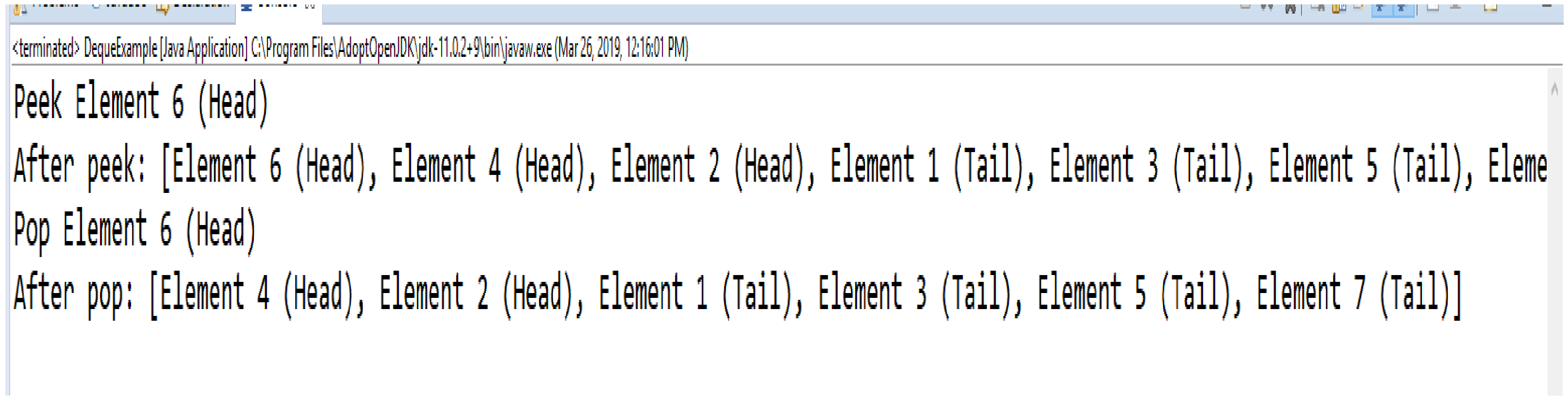
```
//Parcurgere elemente colectie
      System.out.println("Standard Iterator");
      Iterator<String> iterator = deque.iterator();
      while (iterator.hasNext())
      System.out.println("\t" +iterator.next());
      // Parcurgere in ordine inversa
      Iterator<String> reverse
      =deque.descendingIterator();
      System.out.println("Reverse Iterator");
      while (reverse.hasNext())
            System.out.println("\t" +reverse.next());
```



```
Problems  @ Javadoc  Declaration  Console
<terminated> DequeExample [Java Application] C:\Program Files\Adc
Reverse Iterator
	Element 7 (Tail)
	Element 5 (Tail)
	Element 3 (Tail)
	Element 1 (Tail)
	Element 2 (Head)
	Element 4 (Head)
	Element 6 (Head)
```

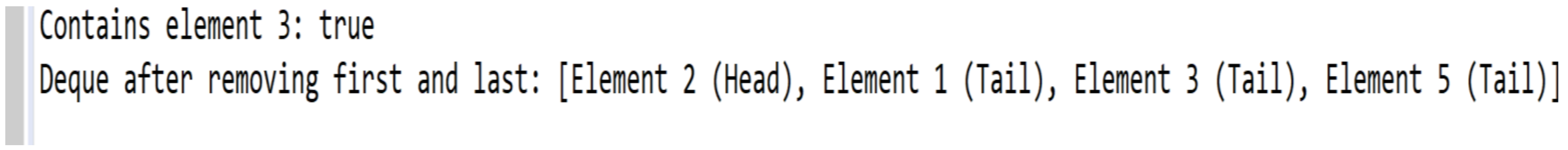
```
//Obtinerea elementului de la varf fara a-l sterge
System.out.println("Peek " + deque.peek());
System.out.println("After peek: " + deque);

//Obtinerea elementului de la varf cu stergere
System.out.println("Pop " + deque.pop());
System.out.println("After pop: " + deque);
```



```
<terminated> DequeExample [Java Application] C:\Program Files\AdoptOpenJDK\jdk-11.0.2+9\bin\javaw.exe (Mar 26, 2019, 12:16:01 PM)
Peek Element 6 (Head)
After peek: [Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Eleme
Pop Element 6 (Head)
After pop: [Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
//Verificarea existentei unui element
System.out.println("Contains element 3: " +
deque.contains("Element 3 (Tail)"));

//Stergere elemente de pe capete
deque.removeFirst();
deque.removeLast();
System.out.println("Deque after removing " +
"first and last: " + deque);
      }
}
```



```
Contains element 3: true
Deque after removing first and last: [Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail)]
```

Stacks

Stacks are *Queues* that have the reverse ordering to the standard *Queue*. Instead of *FIFO* ordering (like a queue – a line at the theater), the ordering of a stack is *Last In First Out* (this can be represented by the acronym ***LIFO***).

-Imagine you have a pile of pancakes. Typically this would be called a "stack" of pancakes because the pancakes are added on top of the previous leaving the most recently added pancake at the top of the stack.

-To remove a pancake, you would have to take off the one that was most recently added – the pancake on the top of the stack.

If you tried to remove the pancake that was added first, you would most likely make a very large mess.

Implementing a Stack - Deque

-One way to implement a *Stack* is by using a Double-Ended Queue (or ***deque***, pronounced "deck", for short – in early versions).

These allow us to insert and remove elements from either end of the queue using methods inside the *Deque* class.

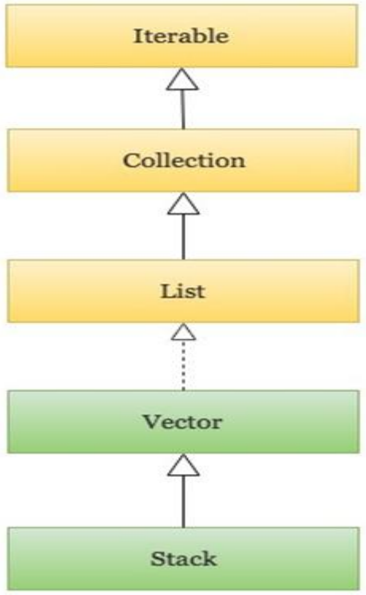
Deque-s like building blocks, allow you to put pieces on the bottom of your structure or on the top, and likewise pull pieces off from the bottom or top.

Deque-s can be implemented by *LinkedLists*.

-Now Java provides the *Stack* class used in implementations.

```
import java.util.Stack;

public class StackExample {
    public static void main(String[ ] args) {
        Stack<String> stackOfCards = new Stack<>( );
        stackOfCards.push("Jack");
        stackOfCards.push("Queen");
        stackOfCards.push("King");
        stackOfCards.push("Ace");
        System.out.println("Stack : " + stackOfCards);
        System.out.println("Is Stack empty? : " + stackOfCards.isEmpty());
        System.out.println("Size of Stack : " + stackOfCards.size());
        /*Search for an element, the search() method returns the 1-based position
of the element from the top of the stack,
         * -1 if the element was not found in the stack*/
        int position = stackOfCards.search("Queen");
        if(position != -1)
 System.out.println("Found the element |"Queen|" at  position : " + position);
        else System.out.println("Element not found");
    }  // main
} // class
```



```
Stack : [Jack, Queen, King, Ace]
Is Stack empty? : false
Size of Stack : 4

Found the element "Queen" at     position : 3
```