UD9. COL·LECCIONS D'ELEMENTS

9. COL·LECCIONS D'ELEMENTS

9.1 Definició i tipus
9.2 Classes genèriques
9.3 Classes històriques
  Enumeration
  Vector. Classes internes
  Hashtable
9.4 Java Collections Framework (JCF)
  Collection
  Iterator i ListIterator
  Comparable i Comparator
  List (ArrayList)
  Queue (LinkedList)
  Set (HashSet)
  SortedSet (TreeSet)
  Map (HashMap)
  ArrayDequeue (Stack)

9.1 Definició i tipus

Una **col·lecció** és una agrupació d'elements (objectes) en la qual s'hi han de poder executar diverses accions: **afegir, recórrer, cercar, extreure**… Tradicionalment, les estructures pensades per a l'organització de la informació s'han classificat segons el tipus d'accés que proporcionen. El llenguatge Java, conscient de la importància d'aquesta organització, proporciona un conjunt complet d'estructures que abraça les diverses possibilitats d'organització de la informació, i constitueix el conegut framework de col·leccions (Java collections framework o JCF).

El JFC és una arquitectura unificada per representar i gestionar col·leccions, independent dels detalls de la implementació que està format per tres tipus de components:

- **Interfícies**. Tipus abstractes de dades (TAD) que defineixen la **funcionalitat** de les col·leccions i funcionalitats de suport.
- **Implementacions**. Classes que implementen les interfícies de les col·leccions, de manera que una interfície pot tenir més d'una implementació (**classe** que la implementi).
- **Algorismes**. Mètodes que efectuen **càlculs** (cerques, ordenacions…) en els objectes de les implementacions.

9.2 Classes genèriques

Les Col·leccions que proporciona Java pertanyen a un tipus de classes especials, anomenades classes genèriques. Les **classes genèriques** són classes que encapsulen dades i mètodes basats en **tipus de dades genèrics** i serveixen de **plantilla** per generar classes a partir de concretar els tipus de dades genèrics. Vegem un exemple d'un problema que podem solucionar

amb les classes genèriques. Imaginem que tenim una classe que simplement el que fa és imprimir el contingut d'un enter:

```
public class MyClassInteger {

	Integer i;

	public MyClassInteger (Integer i) {
		this.i = i;
	}


	public static void main (String[] args) {

		MyClassInteger classInteger = new MyClassInteger(7);
		System.out.println(classInteger.i);
	}
}
```

Si volguéssim ara emprar el mateix programa amb valor tipus Double, hauríem de repetir pràcticament el mateix codi però canviant el tipus de paràmetre:

```
public class MyClassDouble {

	Double d;

	public MyClassDouble (Double d) {
		this.d = d;
	}


	public static void main (String[] args) {

		MyClassDouble classDouble = new MyClassDouble(7.4);
		System.out.println(classDouble.d);
	}
}
```

Per solucionar això podríem emprar objectes tipus Object i fer casts cada vegada segons el tipus de dades a emprar però aquest és un procediment que tendeix a provocar errors. La solució més adient en aquest cas és emprar les classes genèriques. Una classe genèrica es crea afegint un paràmetre a la classe de la següent manera:

```
public class MyClass <T> {

	T obj;

	public MyClass (T obj) {
		this.obj = obj;
	}


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

```
            MyClass<Integer> i = new MyClass<>(7);
            MyClass<Double> d = new MyClass<>(7.4);

            System.out.println(i.obj);
            System.out.println(d.obj);
      }
}
```

Aquí veim que dins el main, cream dues instàncies de la classe MyClass que faran feina amb dos objectes de diferent tipus, per una banda un Integer i per altra un Double.

De la mateixa manera es poden tenir **mètodes genèrics**. Un mètode genèric es crearà de la mateixa manera que un mètode tradicional però sense especificar el tipus de dada que retorna o que rep com a paràmetre. Per exemple:

```
public class GenericMethodClass {
      public <T> void printArray(T[]array) {
            for (T arrayitem : array) {
                  System.out.println(arrayitem);
            }
      }
}
```

Aquest mètode ens permetria imprimir arrays de tipus diferents:

```
public class GenericMethodTest {

      public static void main(String[] args) {
            GenericMethodClass gmc = new GenericMethodClass();

            Integer[] integerArray = {1, 2, 3};
            String[] stringArray = {"Hola", "bona", "tarda"};

            gmc.printArray(integerArray);
            gmc.printArray(stringArray);
      }
}
```

Els paràmetres de les classes genèriques o parametritzades poden ser de qualsevol tipus però sempre han de ser **objectes** (Integer, Double, String, ...). No poden ser dades primitives (~~int, char, double~~, ...)

També podem restringir que no accepti qualsevol tipus de paràmetre sinó que només uns determinats. Això s'aconsegueix amb la paraula **extends** darrera l'identificador del tipus. D'aquesta manera podem indicar que el tipus permesos han de ser només derivats de la classe indicada. Per exemple, si només volem paràmetres numèrics:

```
public class MyClass <T extends Number> {

      T num;

      public MyClass (T num) {
```

```
            this.obj = num;
      }

}
```

Així, els tipus de paràmetres que pot rebre només seran numèrics, per exemple Byte, Double, Float, Integer, Long, Short. Però no acceptarà per exemple String o qualsevol altra tipus que no derivi de Number.

Per una altra banda també podem restringir els tipus de paràmetres com a un concret o el seu tipus superior amb la paraula reservada **super.** Per exemple

```
public class MyClass <T super Integer> {

}
```

acceptaria paràmetres de tipus Integer o de la seva classe 'pare' Number.

De vegades és necessari implementar mètodes que admetin, per paràmetres, objectes de classes genèriques. La sintaxi a utilitzar és utilitzant un ? com a tipus de paràmetre passat:

```
public class GenericClassMethod {
      public static void method (GenericClass<?,?> obj) {
            System.out.println(obj);
      }

      public static void main (String args[]) {
            GenericClass <Integer, Float> object1 =
                        new GenericClass <Integer,Float> (new
Integer(20), new Float(42.45));
            GenericClass <Double, Short> object2 =
                        new GenericClass <Double,Short> (new
Double(4.32), new Short((short) 4));
            method(object1);
            method(object2);
      }
}
```

9.3 Classes històriques

Enumeration

Una enumeració és una classe que només conté una llista finita d'objectes declarats durant la seva creació. Vegem un exemple:

```
class EnumExample1{

      public enum Season { WINTER, SPRING, SUMMER, AUTUMN }

      public static void main(String[] args) {

            for (Season s : Season.values())
                  System.out.println(s);
      }
```

```
}
```

En aquest exemple Season és una classe de tipus col·lecció d'elements enumerats que té els seus propis mètodes. Per exemple podem treure l'índex d'un dels seus elements o el valor d'un dels seus elements.

```
class EnumExample1{

	public enum Season { WINTER, SPRING, SUMMER, AUTUMN }

	public static void main(String[] args) {

		for (Season s : Season.values()){
			System.out.println(s);
		}
		System.out.println("Value of WINTER is:
"+Season.valueOf("WINTER"));
		System.out.println("Index of WINTER is:
"+Season.valueOf("WINTER").ordinal());
		System.out.println("Index of SUMMER is:
"+Season.valueOf("SUMMER").ordinal());

	}
}
```

Vector

https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

Com el seu nom indica, Vector<E> representa una taula de **referències a objectes de tipus "E"** que, a diferència de les taules clàssiques de Java (arrays), **pot créixer i reduir el nombre d'elements**. També permet l'accés als seus elements amb un índex, encara que no permet la utilització dels claudàtors "[]" a diferència de les taules clàssiques de Java. Hi ha molts mètodes, entre els quals cal destacar:

El mètode **capacity()**, que retorna la grandària o el nombre d'elements que pot tenir el vector. És el mètode equivalent a la propietatlength de les taules clàssiques de Java.

El mètode **size()**, que retorna el nombre d'elements que realment conté el vector. Per tant, a diferència de les taules clàssiques de Java, no cal mantenir una variable entera com a comptador del nombre d'elements que conté el vector.

El mètode **get(int n)** retorna la referència que hi ha en la posició indicada per n. Les posicions s'enumeren a partir de zero.

El mètode **add(E obj)** permet afegir la referència obj després del darrer element que hi ha en el vector, i amplia automàticament la grandària del vector si aquest era ple.

El mètode **add(int n, E obj)** permet inserir la referència obj a la posició indicada per n sempre que n>=0 i n⇐size().

Per a un coneixement profund de tots els mètodes, cal fer una ullada a la documentació que acompanya el llenguatge Java. Com a dades membres protegides d'aquesta classe cal conèixer:

- capacityIncrement, que indica l'increment que patirà el vector cada vegada que necessiti créixer.
- elementCount, que conté el nombre de components vàlids del vector.
- elementData[] , que és la taula de referències Object en què realment es desen els elements de l'objecte Vector.

En implementar la interfície List, la classe Vector hereta el mètode iterator(), que retorna una referència a la interfície Iterator, la qual permet fer un recorregut pels diferents elements de l'objecte Vector. Però la interfície Iterator va néixer juntament amb el framework de col·leccions quan la classe Vector ja existia. Per això, la classe Vector té el mètode elements(), que retorna una referència a la interfície Enumeration existent des de la versió 1.0 de Java, amb funcionalitat similar a la de la interfície 'Iterator.

Exemple d'utilització de la classe Vector

Aquest exemple mostra el disseny de la classe VectorValorsNumerics pensada per definir vectors que desin valors numèrics (valors que implementin la classe Number) de manera que, a més de proporcionar els mètodes típics de la classe Vector, proporciona mètodes com el càlcul de la mitjana dels valors continguts, comparació de mitjana, capacitat i nombre d'elements vers altres vectors i implementació del mètode toString().

```
import java.util.*;
public class VectorExample1 {
       public static void main(String args[]) {
          //Create an empty vector with initial capacity 4
          Vector<String> vec = new Vector<String>(4);
          //Adding elements to a vector
          vec.add("Tiger");
          vec.add("Lion");
          vec.add("Dog");
          vec.add("Elephant");
          //Check size and capacity
          System.out.println("Size is: "+vec.size());
          System.out.println("Default capacity is: "+vec.capacity());
          //Display Vector elements
          System.out.println("Vector element is: "+vec);
          vec.addElement("Rat");
          vec.addElement("Cat");
          vec.addElement("Deer");
          //Again check size and capacity after two insertions
          System.out.println("Size after addition: "+vec.size());
          System.out.println("Capacity after addition is: "+vec.capacity()
);
          //Display Vector elements again
```

```
            System.out.println("Elements are: "+vec);
            //Checking if Tiger is present or not in this vector
              if(vec.contains("Tiger"))
              {
                 System.out.println("Tiger is present at the index " +vec.in
dexOf("Tiger"));
              }
              else
              {
                 System.out.println("Tiger is not present in the list.");
              }
              //Get the first element
            System.out.println("The first animal of the vector is = "+vec.fi
rstElement());
            //Get the last element
            System.out.println("The last animal of the vector is = "+vec.las
tElement());
        }
}
```

Exercici 9.1: Modificar el programa de gestió de un institut (exercici 6.5) per a que alumnes i professors s'emmagatzemin dins un vector.

Hashtable

La classe Hashtable<K, V> deriva de la classe abstracta Dictionary i implementa la interfície Map, la interfície Cloneable per poder clonar objectes Hashtable amb el mètode clone(), i la interfície Serializable per poder convertir objectes Hashtable en cadenes de caràcters.

Un objecte Hashtable és una **taula que relaciona una clau amb un element**, utilitzant tècniques Hash, en què **qualsevol objecte** no null pot ser clau i/o element. La classe a què pertanyen les claus ha d'implementar els mètodes **hashCode()** i **equals()** (heretats de la classe Object) per tal de poder fer **cerques** i **comparacions**. El mètode hashCode() ha de retornar un **enter únic i diferent** per cada clau, que sempre és el mateix dins una execució del programa però que pot canviar en diferents execucions. A més, per dues claus que resultin iguals segons el mètode equals(), el mètode hashCode() ha de retornar el mateix valor enter.

Els objectes Hashtable estan dissenyats per mantenir una grup de **parelles clau/element**, de manera que permeten la **inserció** i la **cerca** d'una manera molt eficient i **sense cap tipus d'ordenació**. Cada objecte Hashtable té dues dades membre: "capacity" i "loadFactor" (entre 0.0 i 1.0). Quan el nombre d'elements de l'objecte Hashtable supera el producte "capacity*loadFactor", l'objecte Hashtable creix cridant el mètode rehash(). Un "loadFactor" més gran apura més la memòria però és menys eficient en les cerques. És convenient partir d'una Hashtable suficientment gran per no estar ampliant contínuament.

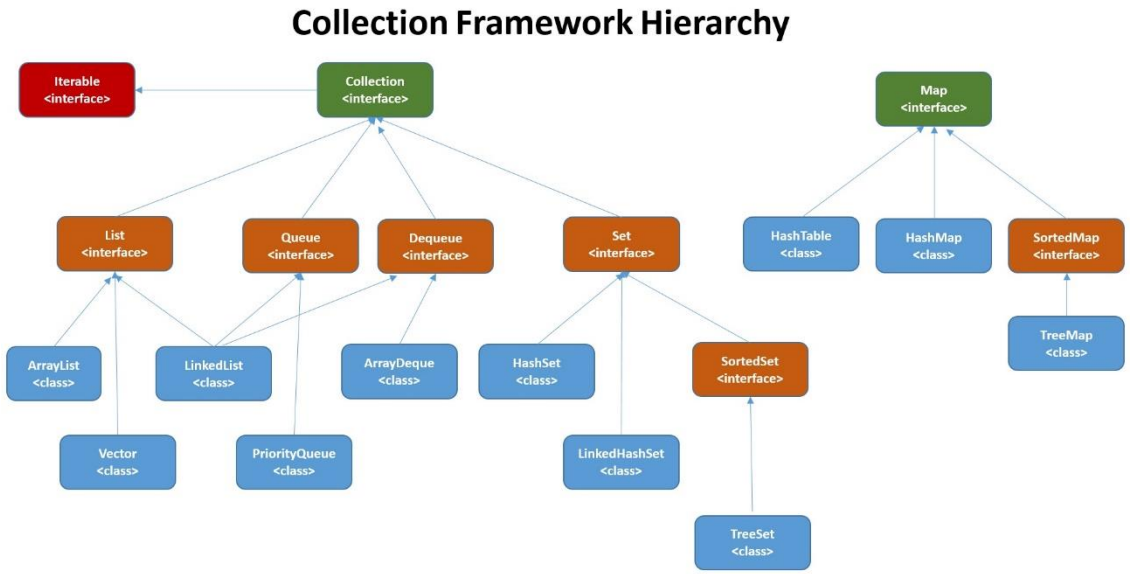
Exemple d'utilització de la classe Hashtable

```
import java.util.*;
public class HashTableExample1{
  public static void main(String args[]){
    HashTable<Integer,String> map=new HashTable<Integer,String>();//Creati
ng HashTable
    map.put(1,"Mango");  //Put elements in Map
    map.put(2,"Apple");
    map.put(3,"Banana");
    map.put(4,"Grapes");

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

Exercici 9.2 : Modificar l'exercici 6.5 anterior per emmagatzemar els alumnes i professors en un HashTable que tengui el DNI com a key.

9.3 Java Collections Framework (JCF)



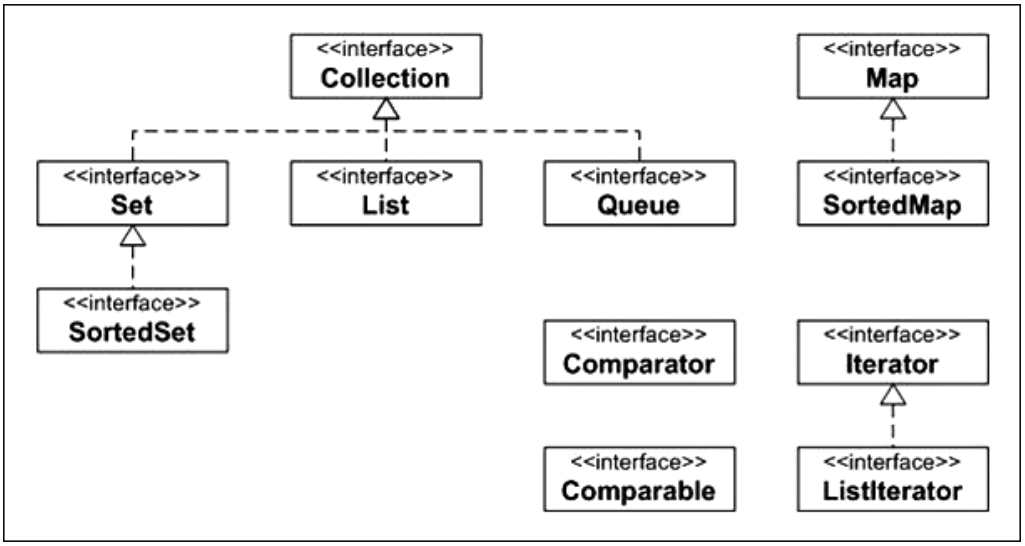
Interfaces

Les interfícies són tipus abstractes de dades (TAD) que defineixen la funcionalitat de les col·leccions i funcionalitats de suport. En el framework de col·leccions cal distingir-ne dos tipus:

- Interfícies que constitueixen el cor del framework i defineixen els diferents tipus de col·leccions: **Collection**, **Set**, **List**, **Queue**, **SortedSet**, **Map** i **SortedMap**, conegudes normalment com interfícies del JCF. Aquest conjunt d'interfícies, al seu torn, està organitzat en dues jerarquies: una, que agrupa les col·leccions amb **accés per posició** (seqüencial i directe) i que té la interfície Collection per arrel, i l'altra, que defineix les col·leccions amb **accés per clau** i que té la interfície Map per arrel.

- Interfícies de suport: **Iterator**, **ListIterator**, **Comparable** i **Comparator**.

L'explicació detallada de cadascuna de les interfícies del framework de col·leccions cal cercar-la en la documentació del llenguatge Java.



Collection

La interfície *Collection<E>* és la interfície pare de la jerarquia de col·leccions amb accés per posició (seqüencial i directe) i comprèn col·leccions de diversos tipus:

- Col·leccions que permeten elements duplicats i col·leccions que no els permeten.

- Col·leccions ordenades i col·leccions desordenades.

- Col·leccions que permeten el valor null i col·leccions que no el permeten.

La seva definició, extreta de la documentació de Java, és força autoexplicativa atesos els noms que utilitzen:

```
public interface Collection<E> extends Iterable<E> {
// Basic operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element);//optional
boolean remove(Object element);//optional
Iterator<E> iterator();

// Bulk operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);//optional
boolean removeAll(Collection<?> c);//optional
boolean retainAll(Collection<?> c);//optional
void clear(); //optional

// Array operations
Object[] toArray();
<T> T[] toArray(T[] a);
}
```

El comentari optional acompanyant un mètode d'una interfície indica que el mètode pot no estar disponible en alguna de les implementacions de la interfície. Això pot passar si un mètode de la interfície no té sentit en una implementació concreta. La implementació ha de definir el mètode, però en ser cridat provoca una excepció UnsupportedOperationException.

El llenguatge Java no proporciona cap classe que implementi directament aquesta interfície, sinó que implementa interfícies derivades d'aquesta. Això no és un impediment per implementar directament, quan convingui, aquesta interfície, però la majoria de vegades n'hi ha prou d'implementar les interfícies derivades o, fins i tot, derivar directament de les implementacions que Java proporciona de les diverses interfícies.

El recorregut pels elements d'una col·lecció es pot efectuar, en principi, gràcies al mètode iterator(), que retorna una referència a la interfície Iterator.

Interfaces Iterator i **ListIterator**

Les interfícies de suport Iterator i ListIterator utilitzades per diversos mètodes de les classes que implementen la interfície Collection o les seves subinterfícies permeten recórrer els elements d'una col·lecció.

La definició de la interfície **Iterator** és:

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

Així, per exemple, la referència que retorna el mètode iterator() de la interfície Collection permet recórrer la col·lecció amb els mètodes next() i hasNext(), i també permet eliminar l'element actual amb el mètode remove() sempre que la col·lecció permeti l'eliminació dels seus elements.

La interfície ListIterator permet recórrer els objectes que implementen la interfície List (subinterfície de Collection) en ambdues direccions (endavant i endarrere) i efectuar algunes modificacions mentre s'efectua el recorregut. Els mètodes que defineix la interfície són:

```java
public interface ListIterator<E> extends Iterator<E>{
  boolean hasNext();
  E next();
  boolean hasPrevious();
  E previous();
  int nextIndex();
  int previousIndex();
  void remove();
  void set(E e);
  void add(E e);
}
```

El recorregut pels elements de la col·lecció (llista) s'efectua amb els mètodes next() i previous(). En una llista amb n elements, els elements es numeren de 0 a n-1, però els valors vàlids per a l'índex iterador són de 0 a n, de manera que l'índex x es troba entre els elements x-1 i x i, per tant, el mètode previousIndex() retorna x-1 i el mètode nextIndex() retorna x; si l'índex és 0, previousIndex() retorna -1, si l'índex és n, nextIndex() retorna el resultat del mètode size().

Interfaces Comparable i comparator

Les interfícies de suport Comparable i Comparator estan orientades a mantenir una **relació d'ordre** en les classes del framework de col·leccions que implementen interfícies que faciliten ordenació (List, SortedSet i SortedMap).

La interfície **Comparable** consta d'un únic mètode:

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

El mètode **compareTo()** compara l'objecte sobre el qual s'aplica el mètode amb l'objecte rebut per paràmetre i retorna un **enter negatiu, zero o positiu** en funció de si l'objecte sobre el qual s'aplica el mètode és menor, igual o major que l'objecte rebut per paràmetre. Si els dos objectes no són comparables, el mètode ha de generar una excepció ClassCastException.

El llenguatge Java proporciona un munt de **classes que implementen aquesta interfície** (String, Character, Date, Byte, Short, Integer,Long, Float, Double, BigDecimal, BigInteger…). Per tant, totes aquestes classes proporcionen una implementació del mètode compareTo().

Tota implementació del mètode compareTo() hauria de ser compatible amb el mètode **equals()**, de manera que compareTo() retorni zero únicament si equals() retorna true i, a més, hauria de verificar la propietat transitiva, com en qualsevol relació d'ordre.

Les col·leccions de classes que implementen la interfície Comparable es poden **ordenar** amb els mètodes **static Collections.sort()** i **Arrays.sort()**.

Exemple d'utilització de la interfície Comparable en la classe Persona

```java
public class Persona implements Comparable {
	private String dni;

	...

	public final int compareTo (Object obj) {
		if (obj instanceof Persona) return
dni.compareTo(((Persona)obj).dni);
		throw new ClassCastException(); // Instrucció que genera una
excepció
	}
```

```
        ...
}
```

Amb aquesta versió de la classe Persona, podem utilitzar el mètode Arrays.sort() per **ordenar una taula de persones** (bé, d'objectes de classes derivades perquè la classe Persona és abstracta). El programa següent ens ho demostraria:

```
import java.util.Arrays;

public class ProvaPersona {
        public static void main(String args[]) {
                Persona t[] = new Persona[6];
                t[0] = new Alumne("99999999","Anna",20);
                t[1] = new Alumne("00000000","Pep",33,'m');
                t[2] = new Alumne("22222222","Maria",40,'s');
                t[3] = new Alumne("66666666","Àngel",22);
                t[4] = new Alumne("11111111","Joanna",25,'M');
                t[5] = new Alumne("55555555","Teresa",30,'S');

                System.out.println("Contingut inicial de la taula:");
                for (int i=0; i<t.length; i++) System.out.println(" "+t[i]);
                Arrays.sort(t);
                System.out.println("Contingut de la taula després d'haver estat
ordenada:");
                for (int i=0; i<t.length; i++) System.out.println(" "+t[i]);
        }
}
```

La interfície **Comparator** permet **ordenar conjunts d'objectes que pertanyen a classes diferents**. Per establir un ordre en aquests casos, el programador ha de subministrar un objecte d'una classe que implementi aquesta interfície:

```
public interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object obj);
}
```

L'objectiu del mètode equals() és **comparar objectes Comparator**. En canvi, l'objectiu del mètode compare() és **comparar dos objectes de diferents classes i obtenir un enter negatiu, zero o positiu**, en funció de si l'objecte rebut per primer paràmetre és menor, igual o major que l'objecte rebut per segon paràmetre.

Exemple d'utilització de la interfície Comparator

La situació que presentem aquí no és gens lògica, però ens serveix per introduir un exemple d'utilització clara de la interfície Comparator. Suposem que tenim una taula que conté objectes de les classes Persona, Date, Double i Short i la volem ordenar. Evidentment, entre aquests elements no hi ha cap ordre natural.

Suposem que es decideix que cal ordenar una tal taula deixant, en primer lloc, els elements Date, seguits dels elements Double, després els elements Persona i, finalment, els elements Short. A més, entre el grup d'elements d'un mateix tipus, es demana que actuï l'ordre natural definit en aquest tipus (definit per la implementació del mètode compareTo() de la interfície Comparable).

El programa següent mostra com es pot dissenyar una classe xComparator que implementa la interfície Comparator i permet definir un objecte que, passat al mètode Arrays.sort(), permet ordenar una taula com la indicada en l'ordre demanat.

//Fitxer ProvaComparator.java

```
import java.util.*;

public class ProvaComparator {
	public static void main (String args[]) {
		Object t[] = new Object[6];
		t[0] = new Alumne("99999999","Anna",20);
		t[1] = new Date();
		t[2] = new Alumne("22222222","Maria",40,'s');
		t[3] = new Double(33.33);
		t[4] = new Short((short)22);
		t[5] = new Date(109,0,1);
		System.out.println("Contingut inicial de la taula:");
		for (int i=0; i<t.length; i++)
			System.out.println(" "+t[i].getClass().getName()+" -
"+t[i]);
		Arrays.sort(t,new MyComparator());
		System.out.println("Contingut de la taula després d'haver estat
ordenada:");
		for (int i=0; i<t.length; i++)
			System.out.println(" "+t[i].getClass().getName()+" -
"+t[i]);
	}
}
```

//Fitxer MyComparator.java

```
public class MyComparator implements Comparator <Object> {
	public int compare (Object o1, Object o2) {
		if (o1 instanceof Date)
			if (o2 instanceof Date) return
((Date)o1).compareTo((Date)o2);
			else return -1;
		else if (o1 instanceof Double)
			if (o2 instanceof Date) return 1;
			else if (o2 instanceof Double) return
((Double)o1).compareTo((Double)o2);
			else return -1;
		else if (o1 instanceof Persona)
			if (o2 instanceof Date || o2 instanceof Double) return 1;
			else if (o2 instanceof Persona) return
((Persona)o1).compareTo((Persona)o2);
			else return -1;
		else if (o1 instanceof Short)
			if (o2 instanceof Short) return
((Short)o1).compareTo((Short)o2);
			else return 1;
		else throw new ClassCastException(); // Instrucció que genera
una excepció
	}
}
```

Interfícies "Set" i "SortedSet"

La interfície Set<E> és destinada a col·leccions que **no mantenen cap ordre d'inserció** i que **no poden tenir dos o més objectes iguals**. Correspon al concepte matemàtic de **conjunt**.

El conjunt de **mètodes** que defineix aquesta interfície és idèntic al conjunt de mètodes definits per la interfície Collection<E>.

Les **implementacions** de la interfície Set<E> necessiten el mètode **equals()** per veure si dos objectes del tipus de la col·lecció són iguals i, per tant, no permetre'n la coexistència dins la col·lecció.

La crida **set.equals(Object obj)** retorna **true** si "obj" també és una instància que implementi la interfície Set, els dos objectes ("set" i "obj") tenen el mateix nombre d'elements i tots els elements d'"obj" estan continguts en set. Per tant, el resultat pot ser true malgrat que set i "obj" siguin de classes diferents però que implementen la interfície Set.

Els mètodes de la interfície Set permeten realitzar les operacions algebraiques **unió**, **intersecció** i **diferència**:

- s1.containsAll(s2) permet saber si "s2" està contingut en "s1".
- s1.addAll(s2) permet convertir s1 en la unió d'"s1" i "s2".
- s1.retainAll(s2) permet convertir "s1" en la intersecció d'"s1" i "s2".
- s1.removeAll(s2) permet convertir "s1" en la diferència d'"s1" i "s2".

La interfície SortedSet derivada de Set afegeix mètodes per permetre gestionar **conjunts ordenats**. La seva definició és:

```
public interface SortedSet<E> extends Set<E> {
  // Range-view
  SortedSet<E> subSet(E fromElement, E toElement);
  SortedSet<E> headSet(E toElement);
  SortedSet<E> tailSet(E fromElement);

  // Endpoints
  E first();
  E last();

  // Comparator access
  Comparator<? super E> comparator();
}
```

La relació d'ordre a aplicar sobre els elements d'un objecte col·lecció que implementi la interfície SortedSet es defineix en el moment de la seva construcció, indicant una referència a un objecte que implementi la interfície Comparator. En cas de no indicar cap referència, els elements de l'objecte es comparen amb l'ordre natural.

El mètode comparator() retorna una referència a l'objecte Comparator que defineix l'ordre dels elements del mètode, i retorna null si es tracta de l'ordre natural.

En un objecte SortedSet, els mètodes iterator(), toArray() i toString() gestionen els elements segons l'ordre establert en l'objecte.

Interfície "List"

La interfície List<E> es destina a col·leccions que **mantenen l'ordre d'inserció** i que **poden tenir elements repetits**. Per aquest motiu, aquesta interfície declara mètodes addicionals (als definits en la interfície Collection) que tenen a veure amb l'ordre i l'accés a elements o interval d'elements:

```
public interface List<E> extends Collection<E> {
  // Positional access
  E get(int index);
  E set(int index, E element);    //optional
  void add(int index, E element); //optional
  E remove(int index);            //optional
  boolean addAll(int index, Collection<? extends E> c); //optional

  // Search
  int indexOf(Object o);
  int lastIndexOf(Object o);

  // Iteration
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);

  // Range-view
  List<E> subList(int from, int to);
}
```

La crida list.equals(Object obj) retorna **true** si "obj" també és una instància que implementa la interfície List, i els dos objectes tenen el **mateix nombre d'elements** i contenen **elements iguals** i **en el mateix ordre**. Per tant, el resultat pot ser true malgrat que "list" i "obj" siguin de classes diferents però que implementen la interfície List.

El mètode add(E o) definit en la interfície Collection afegeix l'element "o" **pel final** de la llista, i el mètode remove(Object o) definit en la interfície Collection **elimina la primera aparició** de l'objecte indicat.

Interface "Queue"

La interfície Queue<E> es destina a gestionar col·leccions que **guarden múltiples elements** abans de ser processats i, per aquest motiu, afegeix els mètodes següents als definits en la interfície Collection:

```
public interface Queue<E> extends Collection<E> {
  E element();
  boolean offer(E e);
  E peek();
  E poll();
  E remove();
}
```

En informàtica, quan es parla de cues, es pensa sempre en una gestió dels elements amb un algorisme FIFO (first input, first output -primer en entrar, primer en sortir-), però això no és obligatòriament així en les classes que implementen la interfície Queue. Un exemple són les **cues amb prioritat**, en què els elements s'ordenen segons un valor per a cada element. Per tant, les implementacions d'aquesta interfície han de definir l'ordre dels seus elements si no es tracta d'implementacions FIFO.

Els mètodes típics en la gestió de cues (**encuar**, **desencuar** i **inici**) prenen, en la interfície Queue, dues formes segons la reacció davant una fallada en l'operació: mètodes que retornen una excepció i mètodes que retornen un valor especial (null o false, segons l'operació). La taula ens mostra la classificació dels mètodes segons aquests criteris.

Operació	Mètode que provoca excepció	Mètode que retorna un valor especial
Encuar	boolean add (E e)	boolean offer(E e)
Desencuar	E remove()	E poll()
Inici	E element()	E peek()

Interfaces "Map" i "SortedMap"

La interfície Map<E> es destina a gestionar **agrupacions d'elements als quals s'accedeix mitjançant una clau**, la qual ha de ser **única** per als diferents elements de l'agrupació. La definició és:

```
public interface Map<K,V> {

  // Basic operations
  V put(K key, V value);   //optional
  V get(Object key);
  V remove(Object key);    //optional
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  int size();
  boolean isEmpty();

  // Bulk operations
  void putAll(Map<? extends K, ? extends V> m);  // optional
  void clear();                                  // optional

  // Collection Views
  public Set<K> keySet();
  public Collection<V> values();
  public Set<Map.Entry<K,V>> entrySet();

  // Interface for entrySet elements
  public interface Entry {
    K getKey();
    V getValue();
    V setValue(V value);
  }
}
```

Molts d'aquests mètodes tenen un significat evident, però altres no tant.

El mètode **entrySet()** retorna una visió del Map com a Set. Els elements d'aquest Set són referències a la interfície Map.Entry que és una interfície interna de Map que permet modificar i eliminar elements del Map, però no afegir-hi nous elements. El mètode **get(key)** permet obtenir l'element a partir de la clau. El mètode **keySet()** retorna una visió de les claus com a Set. El mètode **values()** retorna una visió dels elements del Map com a Collection (perquè hi pot haver elements repetits i com a Set això no seria factible). El mètode **put()** permet afegir una parella clau/element mentre que **putAll(map)** permet afegir-hi totes les parelles d'un Map passat per paràmetre (les parelles amb clau nova s'afegeixen i, en les parelles amb clau ja existent en el Map, els elements nous substitueixen els elements existents). El mètode **remove(key)** elimina una parella clau/element a partir de la clau.

La crida map.equals(Object obj) retorna **true** si "obj" també és una instància que implementa la interfície Map i els dos objectes representen el mateix mapatge o, dit amb altres paraules, si l'expressió:

map.entrySet().equals(((Map) obj).entrySet())

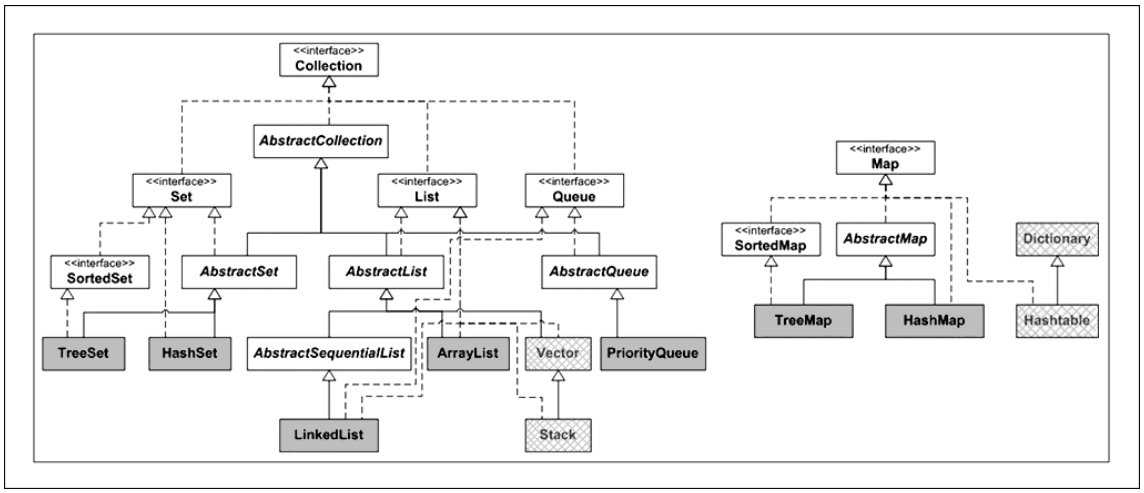
retorna true. Per tant, el resultat pot ser true malgrat que "map" i "obj" siguin de classes diferents però que implementen la interfície Map.

La interfície SortedMap és una interfície Map que permet mantenir **ordenades** les seves parelles en ordre **ascendent segons el valor de la clau**, seguint l'ordre natural o la relació d'ordre proporcionada per un objecte que implementi la interfície Comparator proporcionada en el moment de creació de la instància.

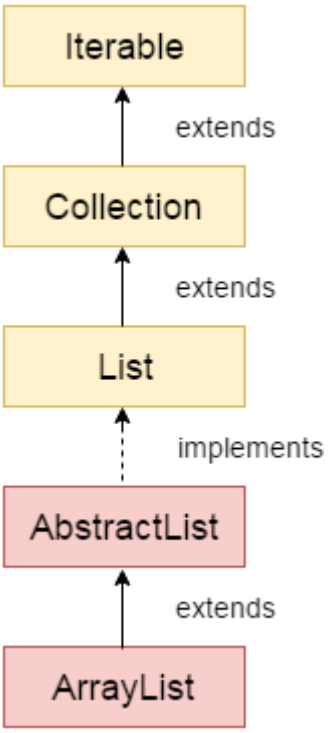
La seva definició, a partir de la interfície Map, és similar a la definició de la interfície SortedSet a partir de la interfície Set:

```
public interface SortedMap<K, V> extends Map<K, V> {
  // Range-view
  SortedMap<K, V> subMap(K fromKey, K toKey);
  SortedMap<K, V> headMap(K toKey);
  SortedMap<K, V> tailMap(K fromKey);
  // Endpoints
  K firstKey();
  K lastKey();
  // Comparator access
  Comparator<? super K> comparator();
}
```

Implementacions



ArrayList



La classe **ArrayList** utilitza un array dinàmic per emmagatzemar els elements. És com un array, però no hi ha límit de mida. Podem afegir o eliminar elements en qualsevol moment. Per tant, és molt més flexible que un array tradicional.

Els punts importants sobre la classe Java ArrayList són:

- Hereta la classe AbstractList i implementa la interfície List.
- Pot contenir elements duplicats.
- Manté l'ordre d'inserció.
- Permet l'accés aleatori perquè la matriu funciona sobre una base d'índex.
- La manipulació és una mica més lenta que la LinkedList perquè s'han de produir molts canvis si s'elimina algun element de la ArrayList.
- No podem crear una ArrayList dels tipus primitius, com ara int, float, char, etc. En aquests casos cal utilitzar la classe d'embolcall requerida (Integer, Float...).
- Java ArrayList s'inicia per la mida. La mida és dinàmica a la ArrayList, que varia segons els elements que s'afegeixen o s'eliminen de la llista.

Exemples: https://www.w3schools.com/java/java_arraylist.asp

Exercici 9.3: Escriu un programa en Java que treballi amb un ArrayList de números enters. El programa ha de realitzar les següents operacions:

1. Crear un ArrayList anomenat numeros.
2. Afegir els següents números a l'ArrayList: 10, 20, 30, 40, 50.

3. Imprimir la llista original.
4. Afegir el número 60 a l'ArrayList.
5. Imprimir la llista després d'afegir el número 60.
6. Eliminar el número 30 de l'ArrayList.
7. Imprimir la llista després d'eliminar el número 30.
8. Accedir al primer element de l'ArrayList i mostrar-lo per pantalla.
9. Modificar el segon element de l'ArrayList canviant-lo per 25.
10. Imprimir la llista després de modificar el segon element.
11. Mostrar per pantalla la quantitat d'elements que té l'ArrayList.
12. Verificar si el número 40 està present a l'ArrayList i imprimir el resultat.
13. Eliminar tots els elements de l'ArrayList.
14. Verificar si l'ArrayList està buit i imprimir el resultat.

Exercici 9.4: Escriu un programa en Java que treballi amb un ArrayList d'objectes de tipus Persona. Cada objecte Persona ha de tenir els següents atributs: nom, DNI i edat.

El programa ha de realitzar les següents operacions:

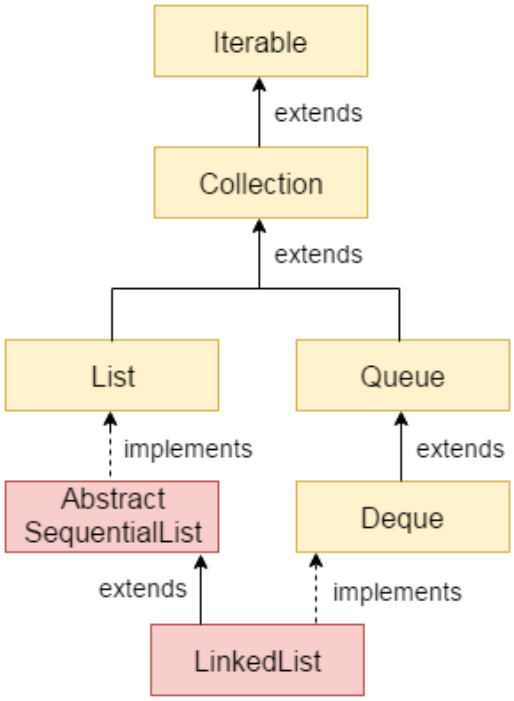
1. Crear un ArrayList anomenat persones.
2. Afegir almenys tres objectes Persona a l'ArrayList, amb diferents noms, DNIs i edats.
3. Mostrar per pantalla la llista original de persones amb els seus detalls (nom, DNI i edat).
4. Afegir una nova persona a l'ArrayList.
5. Mostrar per pantalla la llista actualitzada de persones després d'afegir la nova persona.
6. Eliminar una persona de l'ArrayList utilitzant el seu DNI com a referència.
7. Mostrar per pantalla la llista de persones després d'eliminar la persona especificada.

Exercici 9.5 (Opcional): Pràctica completa de manipulació de dades amb ArrayList:

1. Escriviu un programa Java per crear una ArrayList i afegiu alguns colors (tipus String) i imprimiu la col·lecció.
2. Iterar per tots els elements.
3. Inserir un element a la ArrayList a la primera posició.
4. Recuperar un element (en un índex especificat) d'una ArrayList determinada.
5. Actualitzar un element de matriu específic per un altra element donat.
6. Eliminar el tercer element d'una ArrayList.
7. Cercar un element en una ArrayList.
8. Ordenar una ArrayList determinada.

9. Copiar una ArrayList en una altra. Canviar de posició automàticament (barrejar) elements d'una ArrayList.
10. Invertir els elements d'una ArrayList.
11. Extreure una part d'una ArrayList.
12. Comparar dues ArrayLists
13. Intercanviau dos elements en una ArrayList.
14. Unir dues ArrayLists
15. Clonar una ArrayList en una altra ArrayList.
16. Buidar una ArrayList.
17. Comprovar si una ArrayList està buida o no.
18. Retallar la capacitat d'una ArrayList a la mida de la llista actual.
19. Augmentar la mida d'una ArrayList.
20. Substituir el segon element d'una ArrayList per l'element especificat.
21. Imprimir tots els elements d'una ArrayList utilitzant la posició dels elements.

LinkedList



La classe Java **LinkedList** utilitza una llista doblement enllaçada per emmagatzemar els elements. Proporciona una estructura de dades de llista enllaçada. Hereta la classe AbstractList i implementa les interfícies List i Deque.

Els punts importants sobre Java LinkedList són:

- Pot contenir elements duplicats.
- Manté l'ordre d'inserció.
- La manipulació és ràpida perquè no cal que es produeixi cap canvi.
- Es pot utilitzar com a llista, pila o cua.

Exemples: https://www.w3schools.com/java/java_linkedlist.asp

Els mètodes més interessants que li donen aquesta flexibilitat per muntar tant llistes com piles com cues son:

- addFirst(E e)
- addLast(E e)

- descendingIterator(): Retorna un *iterator* dels elements elements de la cua deque en ordre invers.
- getFirst() : Retorna el primer element de la llista.
- getLast() : Retorna el darrer element de la llista.
- listIterator(int index) : Retorna un iterator dels elements de la llista en l'ordre natural, començant per una posició especificada a *index*.
- offer(E e) : Afegeix l'element al final (darrer element) de la llista.
- offerFirst(E e) : Afegeix l'element al principi (primer element) de la llista.
- offerLast(E e) : Inserta l'element al final de la llista.
- peek() : Recupera, però no elimina, el primer element de la llista.
- peekFirst() : Recupera, però no elimina, el primer element de la llista, o retorna null si la llista està buida.
- peekLast() : Recupera, però no elimina, el darrer element de la llista, o returna null si la llista està buida.
- poll() : Recupera i elimina el primer element de la llista.
- pollFirst() : Recupera i elimina el primer element de la llista o retorna null si la llista està buida.
- pollLast() : Recupera i elimina el darrer element de la llista o retorna null si la llista està buida.
- pop() : Extreu el primer element d'una pila representada per aquest tipus de llista.
- push(E e) : Afageix un element com a primer element d'una pila representada per aquest tipus de llista.

ArrayList vs. LinkedList

Tot i que la classe ArrayList i la classe LinkedList es poden utilitzar de la mateixa manera, es construeixen de manera molt diferent.

Com funciona ArrayList

La classe ArrayList té un array normal al seu interior. Quan s'afegeix un element, es col·loca a l'array. Si l'array no és prou gran, es crea una nova matriu més gran per substituir l'antic i s'elimina l'antic.

Com funciona la LinkedList

LinkedList emmagatzema els seus articles en "contenidors" o "*buckets*". La llista té un enllaç al primer contenidor i cada contenidor té un enllaç al següent contenidor de la llista. Per afegir un element a la llista, l'element es col·loca en un contenidor nou i aquest contenidor s'enllaça amb un dels altres contenidors de la llista.

Quan utilitzar un o l'altra

Utilitzeu una **ArrayList** per emmagatzemar i accedir a les dades, i **LinkedList** quan la major part del temps hagueu de manipular dades.

Exercici 9.6: Escriu un programa en Java que treballi amb una LinkedList de ciutats. El programa ha de realitzar les següents operacions:

1. Crear una LinkedList anomenada llistaCiutats.
2. Afegir les següents ciutats a la llista: Barcelona, Madrid, València, Sevilla.
3. Mostrar per pantalla la llista original de ciutats.
4. Afegir la ciutat Bilbao al principi de la llista.
5. Afegir la ciutat Màlaga al final de la llista.
6. Mostrar per pantalla el primer i l'últim element de la llista.
7. Eliminar la primera ciutat de la llista.
8. Eliminar l'última ciutat de la llista.
9. Mostrar per pantalla la llista de ciutats després de les modificacions.

Exercici 9.7(Opcional): Pràctica de manipulació de dades amb LinkedList

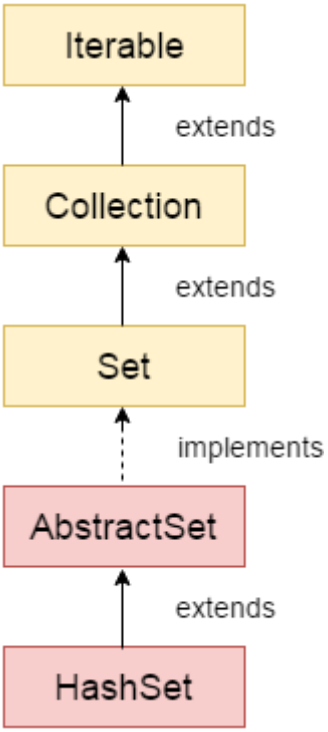
Creau una LinkedList i ompliu-la igual que a l'exercici anterior amb alguns colors (tipus String)

1. Afegiu l'element especificat al final d'una LinkedList.
2. Recorre tots els elements d'una LinkedList.
3. Recorre tots els elements d'una LinkedList començant per la posició especificada.
4. Itera una LinkedList en ordre invers.
5. Inseriu l'element especificat a la posició especificada de la LinkedList.
6. Inseriu elements a la LinkedList a la primera i a l'última posició.
7. Inseriu l'element especificat al capdavant d'una LinkedList.
8. Inseriu l'element especificat al final d'una LinkedList.
9. Inseriu alguns elements a la posició especificada en una LinkedList.
10. Obteniu la primera i l'última ocurrència dels elements especificats en una LinkedList.
11. Mostra els elements i les seves posicions en una LinkedList.
12. Elimina un element especificat d'una LinkedList.
13. Elimina el primer i l'últim element d'una LinkedList.
14. Elimina tots els elements d'una LinkedList.
15. Canvia dos elements en una LinkedList.
16. Barreja els elements d'una LinkedList.
17. Uniu dues LinkedLists.
18. Clona una LinkedList a una altra LinkedList.
19. Elimina i torna el primer element d'una LinkedList.
20. Recupera, però no elimina, el primer element d'una LinkedList.

21. Recupera, però no elimina, l'últim element d'una LinkedList.
22. Comproveu si un element concret existeix en una LinkedList.
23. Converteix una LinkedList en una ArrayList.
24. Compara dues LinkedLists.
25. Comprovar que una LinkedList està buida o no.
26. Substituïu un element d'una LinkedList.

Exercici 9.8: Modificar l'exercici 6.12 (cua de persones) per a que emmagatzemi la cua de persones dins una LinkedList. Comprovau que el tractament de les dades emmagatzemades dins aquesta estructura és molt més senzill que amb un array normal.

HashSet



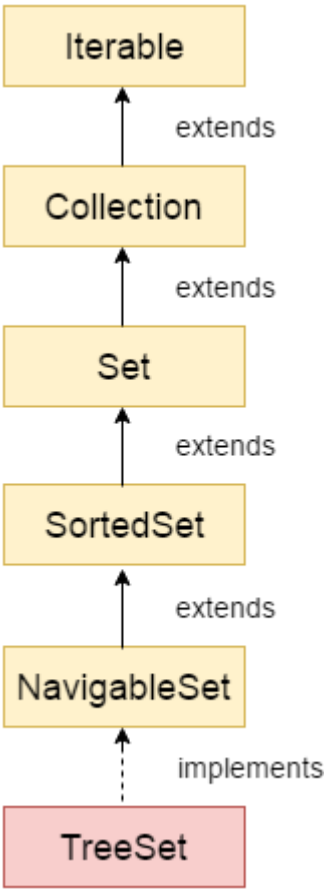
La classe Java HashSet s'utilitza per crear una col·lecció que utilitza una taula hash per a l'emmagatzematge. Hereta la classe AbstractSet i implementa la interfície Set.

Els punts importants sobre la classe Java HashSet són:

- Emmagatzema els elements mitjançant un mecanisme anomenat hashing.
- Només conté elements únics.
- Permet valor nul.
- No manté l'ordre d'inserció. Aquí, els elements s'insereixen en funció del seu codi hash.
- És el millor enfocament per a les operacions de cerca.
- La capacitat inicial predeterminada de HashSet és 16 i el factor de càrrega és 0,75.

Exemples: https://www.w3schools.com/java/java_hashset.asp

TreeSet



La classe **TreeSet** implementa la interfície Set que utilitza un arbre per a l'emmagatzematge. Hereta la classe AbstractSet i implementa la interfície NavigableSet. Els objectes de la classe TreeSet s'emmagatzemen en ordre ascendent. És una de les classes més eficients quan volem emmagatzemar una gran quantitat de dades de forma ordenada.

Els punts importants sobre la classe Java TreeSet són:

- Només conté elements únics com HashSet.
- Els temps d'accés i recuperació de classes de TreeSet són bastant ràpids.
- No permet l'element nul.
- Manté l'ordre ascendent.

Exemples: https://www.javatpoint.com/java-treeset

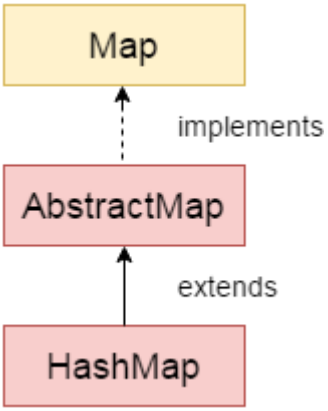
Exercici 9.9: Crea un programa que implementi les següents operacions utilitzant l'estructura de dades HashSet:

- Afegir diferents elements al HashSet (per exemple números: 10, 20, 30, 40, 50).
- Mostrar els elements del HashSet utilitzant un iterador (Iterator). Fixeu-vos en l'ordre en que es mostren.
- Verificar si el nombre 20 o el 35 estan presents al HashSet.
- Eliminar el nombre 40 del HashSet.
- Mostrar els elements del HashSet després d'eliminar l'element.
- Intentar introduït un nombre que ja existeix, per exemple el 50.
- Mostrar els elements del HashSet després d'introduir un element existent. Que ha passat?
- Obtenir la mida del HashSet.
- Netejar el HashSet.
- Verificar si el HashSet està buit.

Exercici 9.10: Crea un programa en Java que implementi les següents operacions utilitzant l'estructura de dades TreeSet:

- Afegir cadenes de text al TreeSet (noms, ciutats, animals, etc).
- Mostrar els elements del TreeSet emprant un iterador. En quin ordre es mostren?
- Verificar si una cadena de text específica està present al TreeSet.
- Intentar afegir un element ja existent. Que ha passat?
- Eliminar una cadena de text del TreeSet.
- Mostrar els elements del TreeSet després d'eliminar una cadena de text.

HashMap



La classe Java **HashMap** implementa la interfície Map que ens permet emmagatzemar la parella de claus i valors, on les claus han de ser **úniques**. Si intenteu inserir la clau duplicada, substituirà l'element de la clau corresponent. És fàcil realitzar operacions utilitzant l'índex de claus com ara l'actualització, la supressió, etc.

HashMap és la versió moderna de la classe original Hashtable heretada. També ens permet emmagatzemar els elements nuls, però només hi hauria d'haver una clau nul·la. Des de Java 5, es denota com HashMap<K,V>, on K significa clau i V per valor. Hereta la classe AbstractMap i implementa la interfície Map.

Punts a recordar:

- Conté valors basats en la clau.
- Només conté claus úniques.
- Pot tenir una clau nul·la i diversos valors nuls.
- No manté cap ordre.
- La capacitat inicial predeterminada de la classe Java HashMap és 16 amb un factor de càrrega de 0,75.

Exemples: https://www.w3schools.com/java/java_hashmap.asp

Exercici 9.11: Implementa un programa que permeti gestionar una llista de contactes utilitzant un HashMap. Basta que per al contacte emmagarzemis només el nom amb un Sting. És a dir que pots fer un objecte del tipus HashMap<String,Integer>.

El programa ha de permetre les següents opcions:

- Mostrar tots els contactes emmagatzemats en el HashMap.
- Afegir un nou contacte, demanant a l'usuari que introdueixi el nom i el número de telèfon.
- Buscar un contacte per nom i mostrar el seu número de telèfon.

Mostra un menú d'opcions a l'usuari i executa la funcionalitat corresponent segons la seva elecció i assegura't de gestionar els casos en què no es trobi cap contacte amb el nom proporcionat durant la cerca.

ArrayDequeue (Stack)

Una altra de les estructures típiques que en qualque moment es poden necessitar son les piles (stack), que son unes estructures LIFO (last in first out) on el darrer element que s'ha introduït és el primer element que s'ha d'extreure. Aquesta estructura és igual a una pila de llibres per exemple que quan en deixam un dalt de tot és el primer que agafarem.

Aquesta estructura s'implementa amb una ArrayDequeue i els mètodes més importats per utilitzarla son push per introduir un element i pop per extreure'l.

Exemple: https://www.geeksforgeeks.org/arraydeque-in-java/

```
import java.util.*;

// ArrayDequeDemo
public class stack {
    public static void main(String[] args)
    {

        Deque<Integer> de_que = new ArrayDeque<Integer>(10);
        de_que.add(10);
        de_que.add(20);
        de_que.add(30);
        de_que.add(40);
        de_que.add(50);
        System.out.println("Initial elements: " + de_que);
        de_que.push(265);
        de_que.push(984);
        de_que.push(2365);
        System.out.println("After pushing 3 elements: " + de_que);
        System.out.println("Element popped: " + de_que.pop());
        System.out.println("Final result: "+ de_que);
    }
}
```

Exercici 9.12: Mastermind amb emmagatzematge de partides.

Modificar el programa del Mastermind perquè emmagatzemi les partides jugades dins d'una Collection de les que hem vist i que s'adapti millor al que volem emmagatzemar. Les dades que ha de tenir de cada partida son: nom del jugador, seqüència d'intents que fa el jugador, les respostes que dona l'ordinador i la puntuació final. La puntuació de cada tirada s'aconseguirà sumant 2 punts per cada color encertat a la seva posició i 1 punt per color encertat però mal posicionat. Quan el jugador encerta la combinació es multiplicarà la puntuació que tenia fins aquell moment pel número d'intents que encara li quedaven.

Per exemple si a una tirada l'ordinador respon 2 encertats i 1 mal posicionat, la puntuació d'aquesta tirada serà 2*2 + 1*1 = 5 punts.

Exercici 9.13 (opcional): (Pràctica amb diferents elements de col·leccions)

Una llibreria necessita un sistema per gestionar el seu inventari i les transaccions de compra de llibres. El sistema ha de ser capaç de mostrar l'inventari de llibres disponibles, permetre als clients realitzar compres i actualitzar l'inventari després d'una compra exitosa.

Desenvolupa un programa que compleixi els següents requisits:

- Crear una llista de llibres disponibles a la llibreria. Utilitzant un objecte tipus List.
- Crear un conjunt de clients registrats a la llibreria. Utilitzant un objecte tipus Set.

- Utilitzar un mapa per emmagatzemar l'inventari de llibres i la seva quantitat disponible.
- Mostrar l'inventari de llibres disponibles al principi del programa.
- Permetre als clients registrats (pot ser simplement per nom) realitzar compres especificant el nom del llibre i la quantitat desitjada. Si el client no està registrat se li haurà de donar l'opció de registrar-se.
- Verificar si el llibre està disponible a l'inventari i si hi ha suficient quantitat per a la compra.
- Actualitzar l'inventari després d'una compra exitosa.
- Mostrar un missatge indicant si la compra s'ha realitzat amb èxit o si el llibre no està disponible en la quantitat requerida.
- Mostrar l'inventari actualitzat de llibres disponibles després d'una compra.