

# Programació

## UT8.3. Col·leccions



# Introducció

- Una col·lecció és una estructura de dades organitzada de manera que no només emmagatzema informació sinó que també proveeix d'operacions per accedir i manipular-la.
- En POO a eses estructures de dades les coneixem com a **contenidors d'objectes**. Concretament en Java anomenats **col·leccions (Collections)**.
- La classe que gestiona estos objectes disposa de mètodes per realitzar operacions tals com la cerca, inserció, esborrat d'elements...



# Col·leccions


- Fins ara hem vist una **col·lecció de dades** com és `ArrayList`, però Java proveeix de moltes altres que podran ser utilitzades en funció del problema a resoldre.
- Anem a veure com es gestionen les **l·listes, piles, cues, cues de prioritat, conjunts** o les **taules hash**.



# Collections



*Java Collection Framework* treballa amb dos tipus de contenidors:

- Els que **emmagatzemen una col·lecció d'elements** (collections)
  - Els que **emmagatzemen una col·lecció de parelles clau/valor** (maps)
- 



# Tipus de Collections a Java

- **Set** (conjunt): emmagatzema un grup d'elements no duplicats
- **List** (lista): emmagatzema una col·lecció ordenada d'elements
- **Stack** (pila): emmagatzema els elements seguint el criteri LIFO
- **Queue** (cua): emmagatzema els elements seguint el criteri FIFO
- **PriorityQueue** (cua de prioritat): emmagatzema els elements segons un ordre de prioritat dels seus elements.

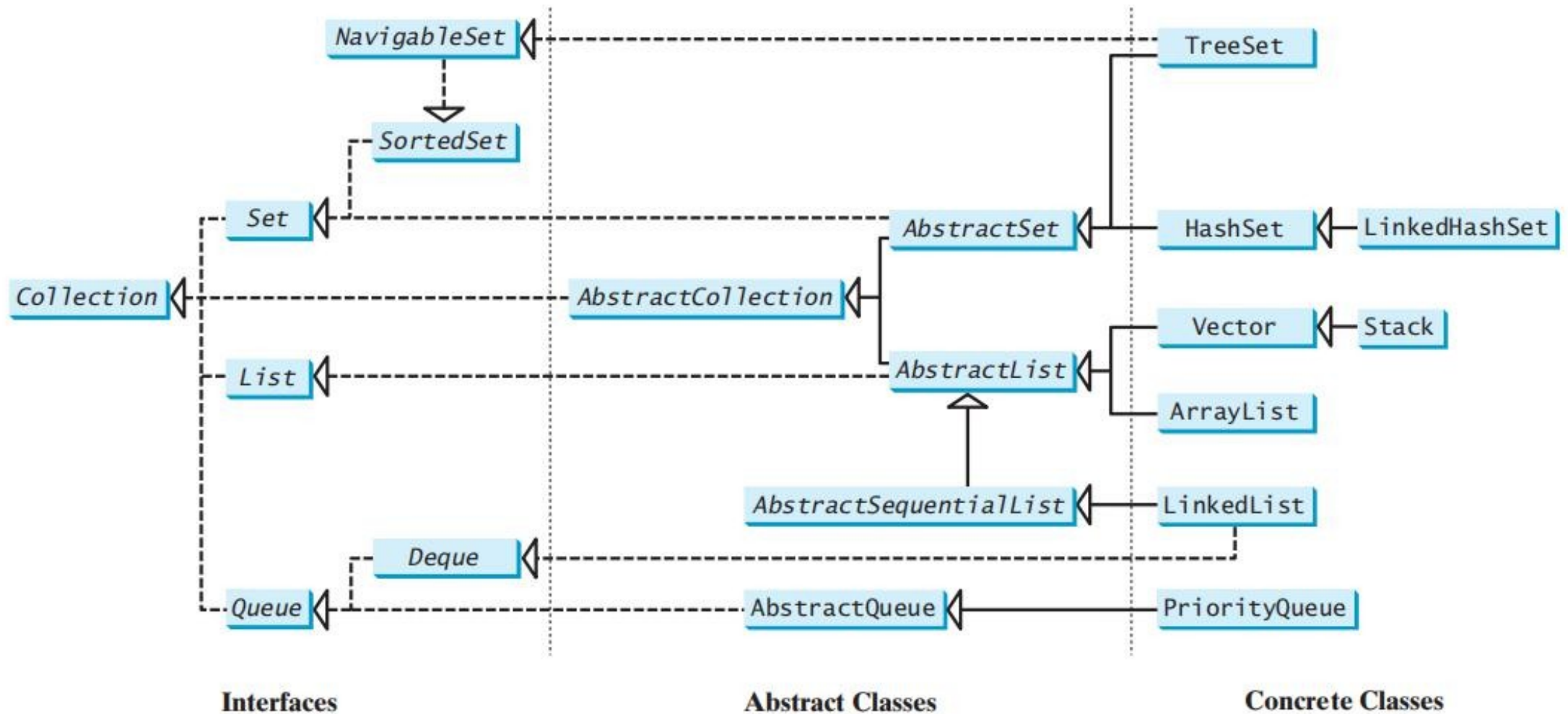


# Collections

- Les operacions comunes d'estes col·leccions estan definides en les interfícies (*Collection*, *Set*, *List*, *Queue...*).
- Les implementacions estan realitzades en les classes concretes que les implementen (*ArrayList*, *LinkedList*, *PriorityQueue*, *Stack*, *HashSet...*).
- Totes les interfícies i classes de Java Collection Framework estan en el paquet `java.util`.

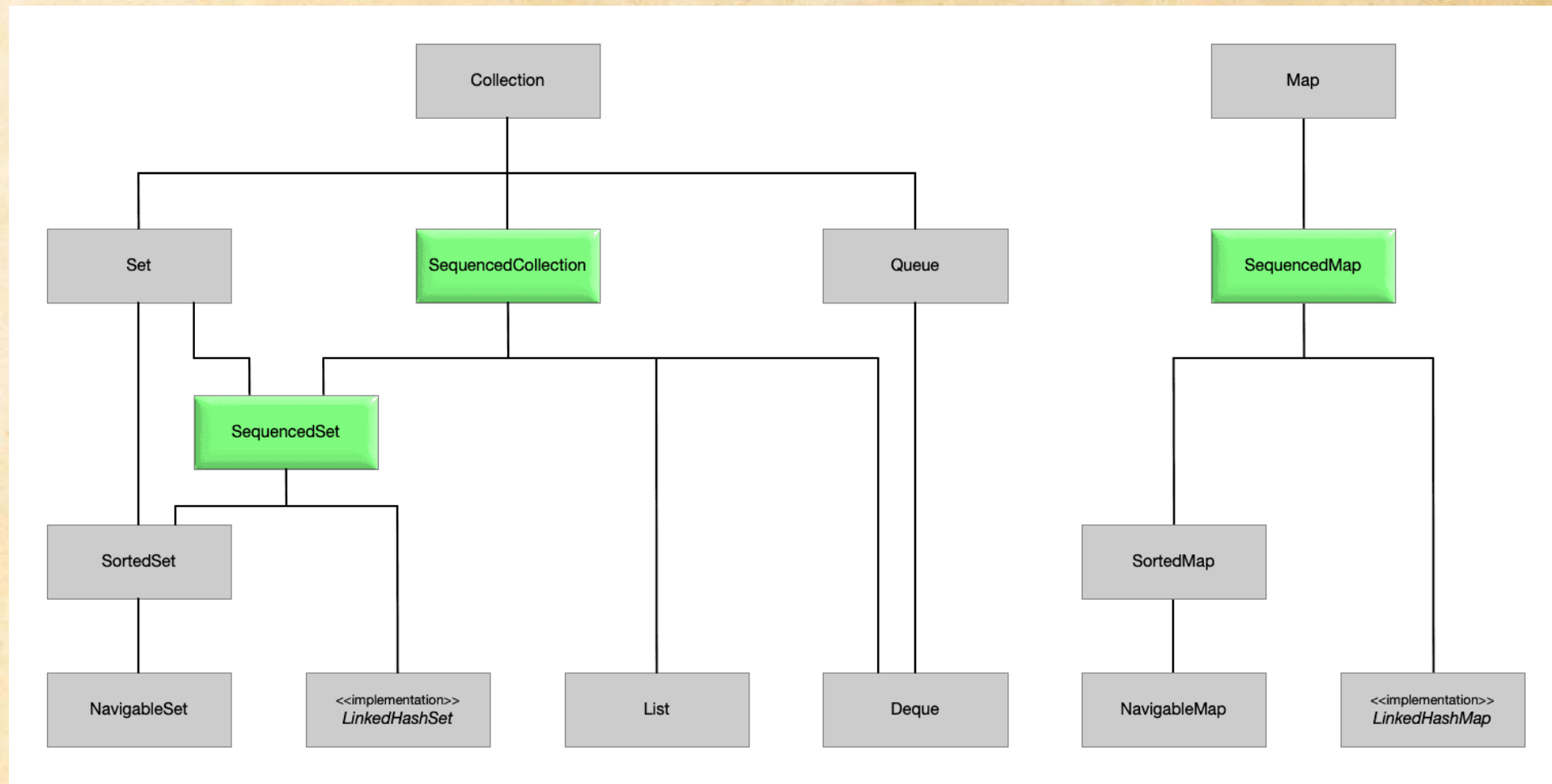


# Java Collections Framework



# Java Collections Framework

*Important: A partir de la versió 21, s'afegeixen noves interfícies:*





# Interfície Iterable

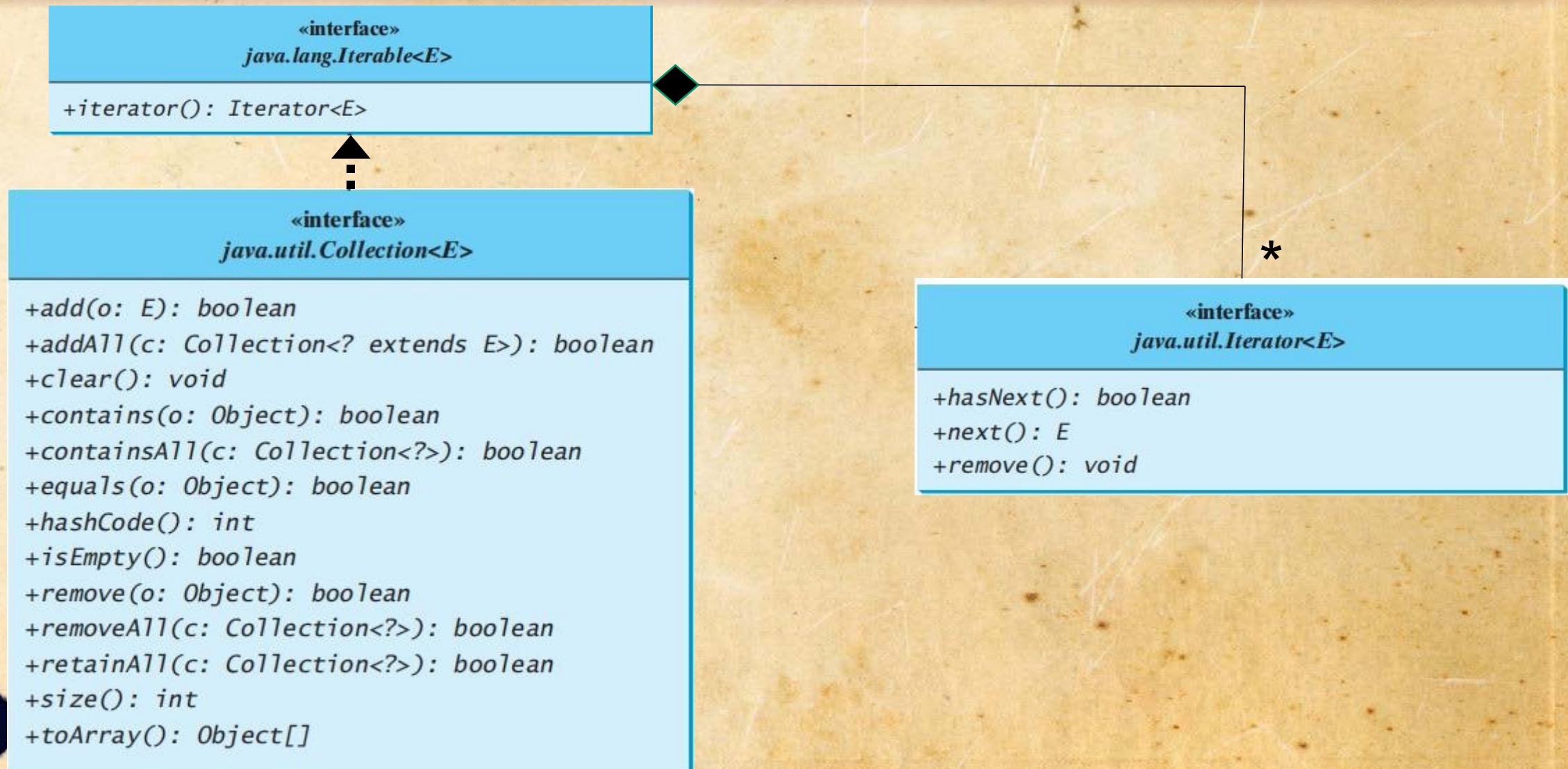
- Tots els Collection, implementen la interfície **Iterable<T>** que obliga a oferir un **iterador** (estructura que permet recórrer els elements de la col·lecció):

```
public interface Iterable<T> {  
  
    Iterator<T> iterator();  
  
}
```

- Qualsevol altre tipus de dades que vulga ser iterable ha d'implementar esta interfície, com per exemple els ArrayList



# Interfície Iterable





# Iterator

- El mètode `iterator()` de la interfície `Iterable` serà accessible en qualsevol col·lecció.
- El objecte `Iterator` que retorna permet recorre els elements d'una col·lecció qualsevol.

```
Integer[] numeros = {1, 2, 4, 5, 1};
ArrayList listaPrueba = new ArrayList(Arrays.asList(numeros));

Iterator<Integer> iterator = listaPrueba.iterator();

while (iterator.hasNext()) {
    Integer numero_i = iterator.next();

    if (numero_i == 1) {
        iterator.remove();
    }
}

System.out.println(listaPrueba.toString());
```



# Tipus de col·leccions: List

- **List** és una interfície, per això no es pot crear un objecte de ella, però altres classes les poden implementar.
- La interfície **List** hereta de la interfície **Collection** i defineix una seqüència d'elements.
- Per a crear una llista, es solen utilitzar les classes concretes **ArrayList** (que ja conéixes) o **LinkedList**



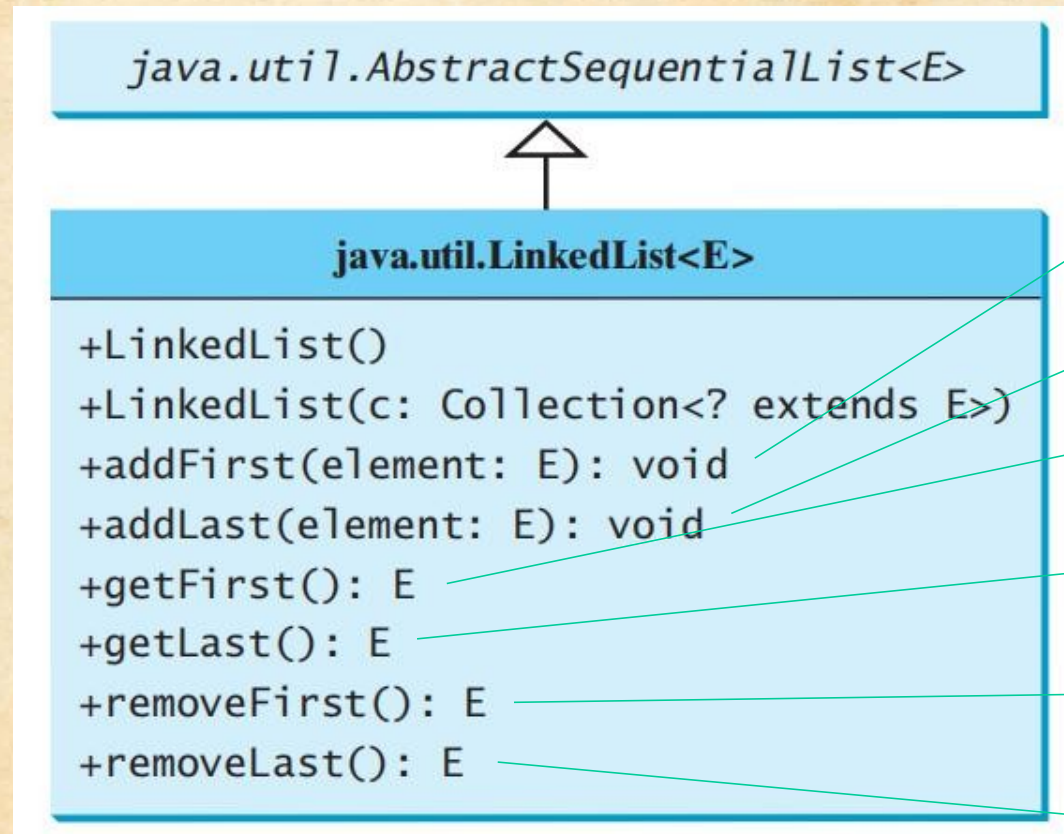
# Mètodes més comuns de List



- Afegir una llista de elements
- Retorna un `ListIterator` (especialització de `Iterator`)
- Retorna una subllista de la general donat un índex des de i un altre fins



# Classe LinkedList

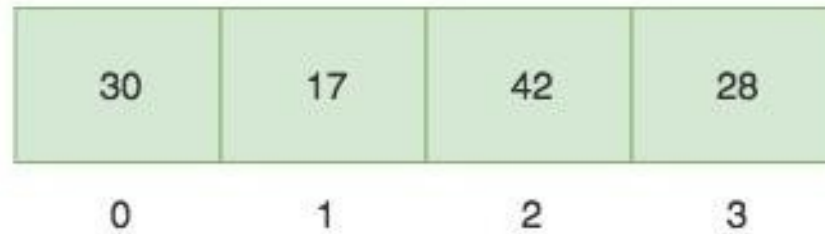


- Afegeix al principi
- Afegeix al final
- Obté el primer element
- Obté l'últim element
- Elimina el primer element
- Elimina l'últim element

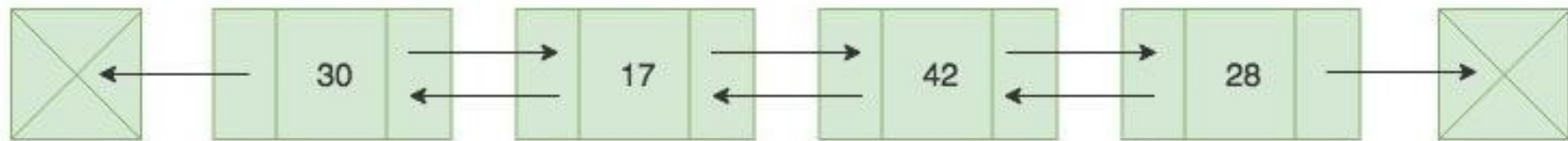


# ArrayList vs LinkedList

Java ArrayList  
Representation



Java LinkedList  
Representation





# Example: LinkedList

```
public static void main(String[] args) {  
    // A LinkedList containing Stock Prices of a company for the last 6 days  
    LinkedList<Double> stockPrices = new LinkedList<>();  
  
    stockPrices.add(45.00);  
    stockPrices.add(51.00);  
    stockPrices.add(62.50);  
    stockPrices.add(42.75);  
    stockPrices.add(36.80);  
    stockPrices.add(68.40);  
  
    // Getting the first element in the LinkedList using getFirst()  
    // The getFirst() method throws NoSuchElementException if the LinkedList is empty  
    Double firstElement = stockPrices.getFirst();  
    System.out.println("Initial Stock Price : " + firstElement);  
  
    // Getting the last element in the LinkedList using getLast()  
    // The getLast() method throws NoSuchElementException if the LinkedList is empty  
    Double lastElement = stockPrices.getLast();  
    System.out.println("Current Stock Price : " + lastElement);  
  
    // Getting the element at a given position in the LinkedList  
    Double stockPriceOn3rdDay = stockPrices.get(2);  
    System.out.println("Stock Price on 3rd Day : " + stockPriceOn3rdDay);  
}
```



# Classe Stack

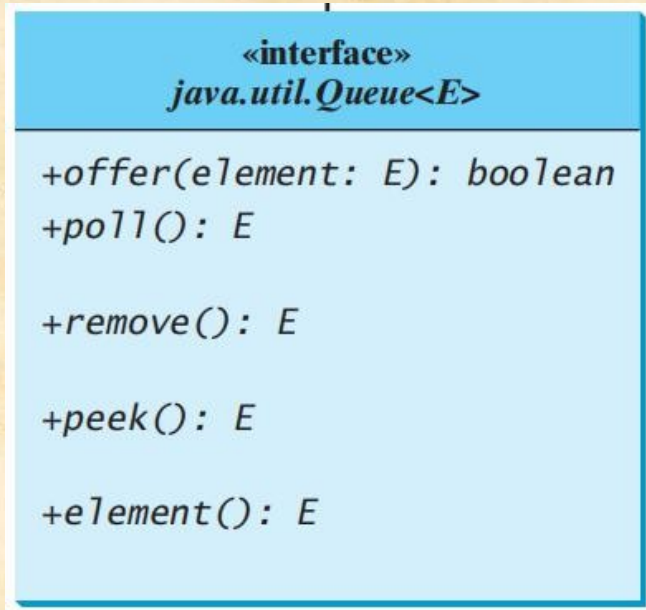
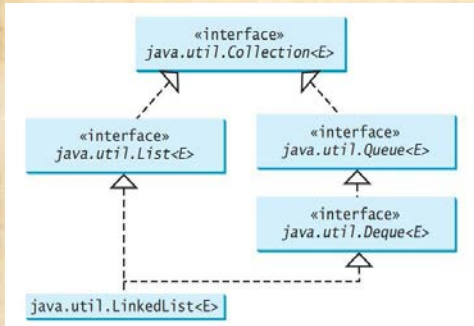
**java.util.Stack<E>**

```
+Stack()  
+empty(): boolean  
+peek(): E  
+pop(): E  
+push(o: E): E  
+search(o: Object): int
```

- **Stack** és una classe instanciable que ens permet gestionar de forma eficient una pila d'elements (Llista LIFO).
- Esta classe hereta de la col·lecció Vector, que no veurem en este curs.
- Vector hereta de List



# Interfície Queue



Summary of Queue methods

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

- Segons el disseny de Java, no hi ha una classe directament instanciable que represente una cua (queue).
- Esta és una interfície que implementen altres classes del *Java Collection Framework* com és **LinkedList** (instanciable)



# Example: Queue

```
public static void main(String[] args) {  
    // Create and initialize a Queue using a LinkedList  
    Queue<String> waitingQueue = new LinkedList<>();  
  
    // Adding new elements to the Queue (The Enqueue operation)  
    waitingQueue.add("Rajeev");  
    waitingQueue.add("Chris");  
    waitingQueue.add("John");  
    waitingQueue.add("Mark");  
    waitingQueue.add("Steven");  
  
    System.out.println("WaitingQueue : " + waitingQueue);  
  
    // Removing an element from the Queue using remove() (The Dequeue operation)  
    // The remove() method throws NoSuchElementException if the Queue is empty  
    String name = waitingQueue.remove();  
    System.out.println("Removed from WaitingQueue : " + name + " | New WaitingQueue : " + waitingQueue);  
  
    // Removing an element from the Queue using poll()  
    // The poll() method is similar to remove() except that it returns null if the Queue is empty.  
    name = waitingQueue.poll();  
    System.out.println("Removed from WaitingQueue : " + name + " | New WaitingQueue : " + waitingQueue);  
}
```

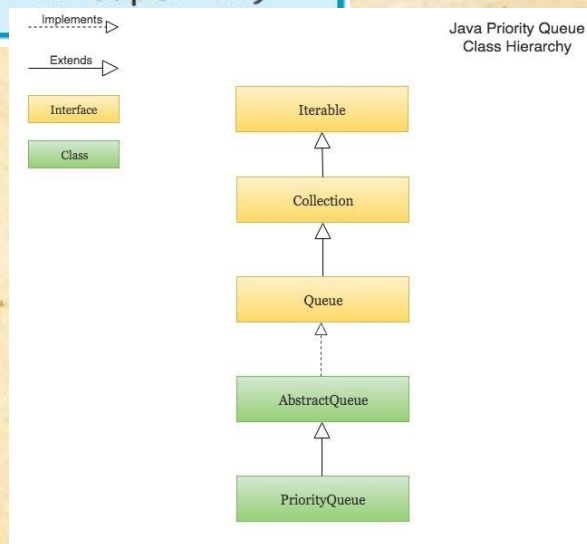
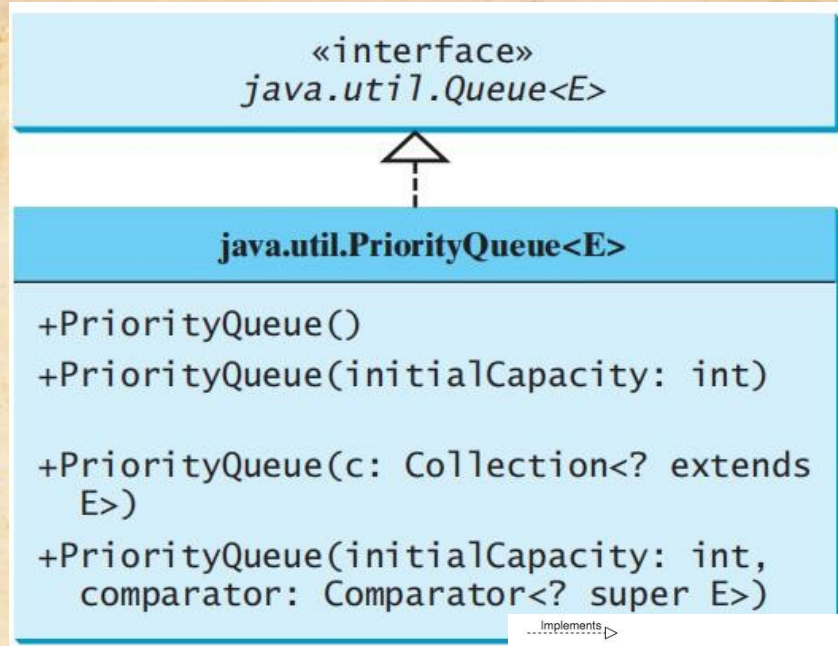
WaitingQueue: [Rajeev, Chris, John, Mark, Steven]

Removed from WaitingQueue : Rajeev | New WaitingQueue: [Chris, John, Mark, Steven]

Removed from WaitingQueue : Chris | New WaitingQueue: [John, Mark, Steven]



# Priority Queue



- Les cues de prioritat, ens permeten inserir elements seguint una prioritat d'inserció determinada.
- Seguiran l'ordre natural dels elements inserits o bé l'ordre definit explícitament a través de la interfície Comparator



# Exemple de Priority Queue

```
import java.util.PriorityQueue;

public class CreatePriorityQueueStringExample {
    public static void main(String[] args) {
        // Create a Priority Queue
        PriorityQueue<String> namePriorityQueue = new PriorityQueue<>();

        // Add items to a Priority Queue (ENQUEUE)
        namePriorityQueue.add("Lisa");
        namePriorityQueue.add("Robert");
        namePriorityQueue.add("John");
        namePriorityQueue.add("Chris");
        namePriorityQueue.add("Angelina");
        namePriorityQueue.add("Joe");

        // Remove items from the Priority Queue (DEQUEUE)
        while (!namePriorityQueue.isEmpty()) {
            System.out.println(namePriorityQueue.remove());
        }
    }
}
```

Ordre naturel  
de String

# Output

Angelina

Chris

Joe

John

Lisa

Robert



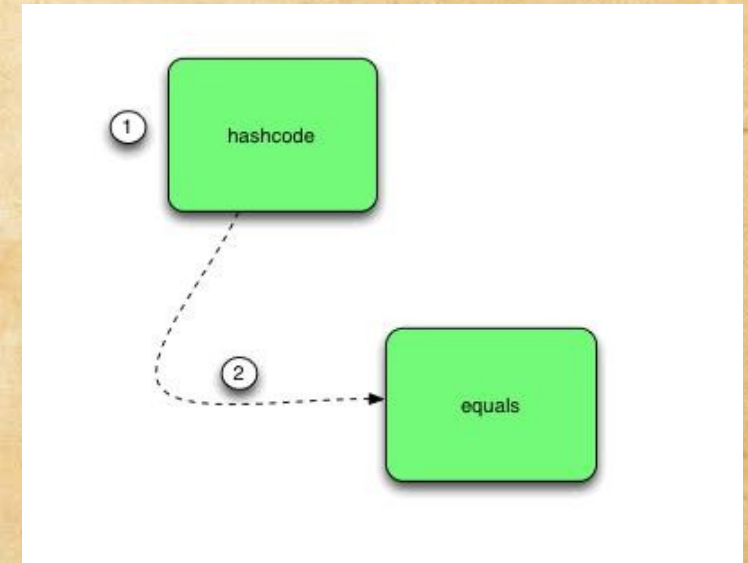
# Set (conjunt)

- Estructura que s'utilitza per a emmagatzemar una llista d'elements no duplicats.
- Set és una interfície que hereta de la interfície `Collection`
- No afegeix mètodes addicionals als que ja proporciona `Collection`, però **assegura la no duplicitat** dels elements afegits.
- Es a dir, no pot haver-hi una parella d'elements `e1`, `e2` que complisquen que `e1.equals(e2)`.



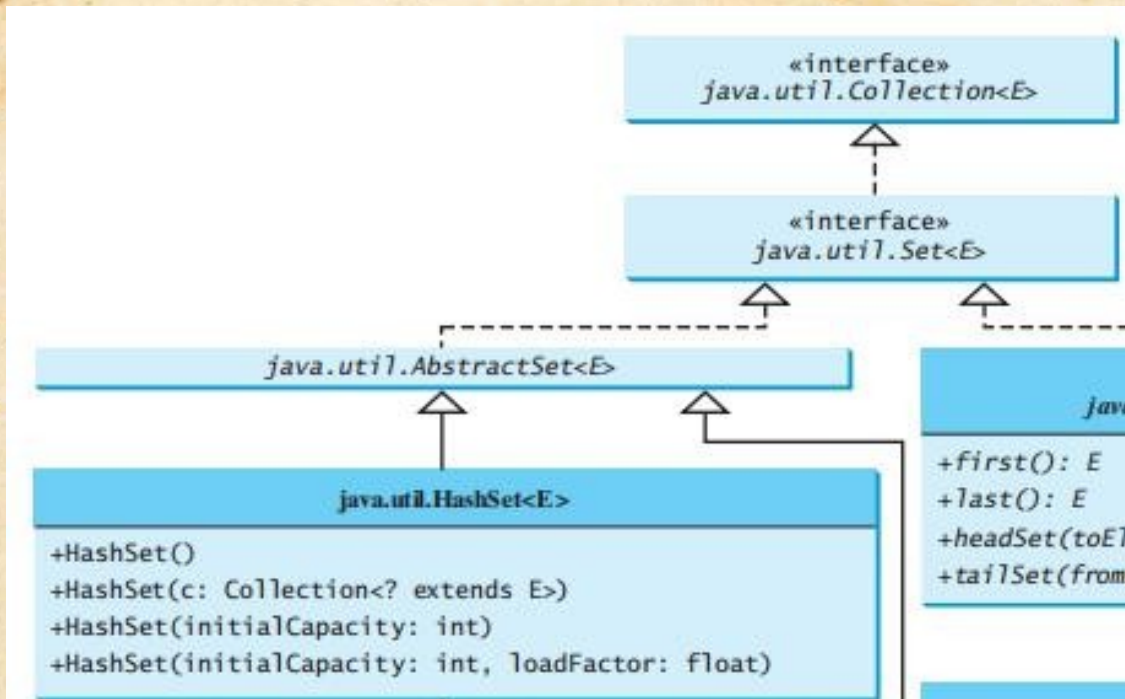
# HashSet

- És una de les classes concretes que implementen Set
- El nom Hash es refereix a que per comparar els elements utilitza el codi hash de cadascun dels elements invocant al mètode hashCode.
- Este mètode, s'invoca en Java cada volta que es comparen dos objectes. S'invoca abans d>equals. Si retorna false, no s'invocarà a equals.





# Exemple de HashSet



```
import java.util.*;
```

```
public class TestHashSet {
    public static void main(String[] args) {
        // Create a hash set
        Set<String> set = new HashSet<>();
```

```
        // Add strings to the set
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Beijing");
        set.add("New York");
```

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

```
        // Display the elements in the hash set
        for (String s: set) {
            System.out.print(s.toUpperCase() + " ");
        }
    }
}
```

No llança excepció, simplement no ho afegeix

[San Francisco, New York, Paris, Beijing, London]  
SAN FRANCISCO NEW YORK PARIS BEIJING LONDON

L'ordre és indeterminat



# LinkedHashSet

- Hereta de HashSet afegint les característiques d'una llista enllaçada.
- Els elements de HashSet no estan ordenats però els de LinkedHashSet sí, ja que poden ser obtinguts per ordre d'inserció

```
public class TestLinkedHashSet {  
    public static void main(String[] args) {  
        // Create a hash set  
        Set<String> set = new LinkedHashSet<>();  
  
        // Add strings to the set  
        set.add("London");  
        set.add("Paris");  
        set.add("New York");  
        set.add("San Francisco");  
        set.add("Beijing");  
        set.add("New York");  
  
        System.out.println(set);  
  
        // Display the elements in the hash set  
        for (Object element: set)  
            System.out.print(element.toLowerCase() + " ");  
    }  
}
```

[London, Paris, New York, San Francisco, Beijing]  
london paris new york san francisco beijing

Ordre  
d'inserció



# TreeSet

- En este tipus de conjunts es pot indicar l'ordre en el que quedaran els elements disposats al conjunt.
- L'ordre el determina la implementació del mètode `compareTo` dels elements introduïts.
- `TreeSet` disposa de mètodes com:
  - `first()`
  - `last()`
  - `lower(dada)`
  - `higher(dada)`
  - `floor(dada)` el més gran que siga menor o igual que la `dada`
  - `ceiling(dada)` el més menut que siga major o igual que la `dada`
  - `pollFirst()` elimina el primer
  - `pollLast()` elimina l'últim



# Exemple de TreeSet

```
public class TestTreeSet {  
    public static void main(String[] args) {  
        // Create a hash set  
        Set<String> set = new HashSet<>();  
  
        // Add strings to the set  
        set.add("London");  
        set.add("Paris");  
        set.add("New York");  
        set.add("San Francisco");  
        set.add("Beijing");  
        set.add("New York");  
  
        TreeSet<String> treeSet = new TreeSet<>(set);  
        System.out.println("Sorted tree set: " + treeSet);  
  
        // Use the methods in SortedSet interface  
        System.out.println("first(): " + treeSet.first());  
        System.out.println("last(): " + treeSet.last());  
        System.out.println("headSet(\"New York\"): " +  
            treeSet.headSet("New York"));  
        System.out.println("tailSet(\"New York\"): " +  
            treeSet.tailSet("New York"));  
  
        // Use the methods in NavigableSet interface  
        System.out.println("lower(\"P\"): " + treeSet.lower("P"));  
        System.out.println("higher(\"P\"): " + treeSet.higher("P"));  
        System.out.println("floor(\"P\"): " + treeSet.floor("P"));  
        System.out.println("ceiling(\"P\"): " + treeSet.ceiling("P"));  
        System.out.println("pollFirst(): " + treeSet.pollFirst());  
        System.out.println("pollLast(): " + treeSet.pollLast());  
        System.out.println("New tree set: " + treeSet);  
    }  
}
```

Ordre de  
compareTo de  
String

Sorted tree set: [Beijing, London, New York, Paris, San Francisco]  
first(): Beijing  
last(): San Francisco  
headSet("New York"): [Beijing, London]  
tailSet("New York"): [New York, Paris, San Francisco]  
lower("P"): New York  
higher("P"): Paris  
floor("P"): New York  
ceiling("P"): Paris  
pollFirst(): Beijing  
pollLast(): San Francisco  
New tree set: [London, New York, Paris]



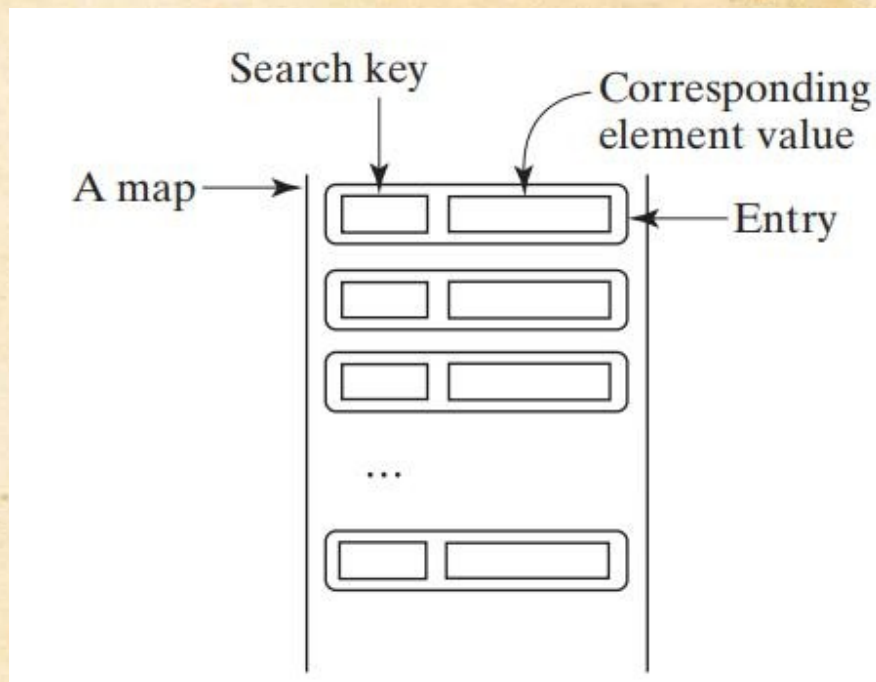
# Map

- És un contenidor que emmagatzema una col·lecció de **parelles clau/valor**.
- Les claus fan la funció dels **índexs** d'un array.
- Als arrays els índexs són enters, i als Map les claus poden ser **qualsevol tipus de objecte**.
- Un Map **no pot contindre claus duplicades**.
- En altres llenguatges es coneix com “**array associatiu**” o “hash”

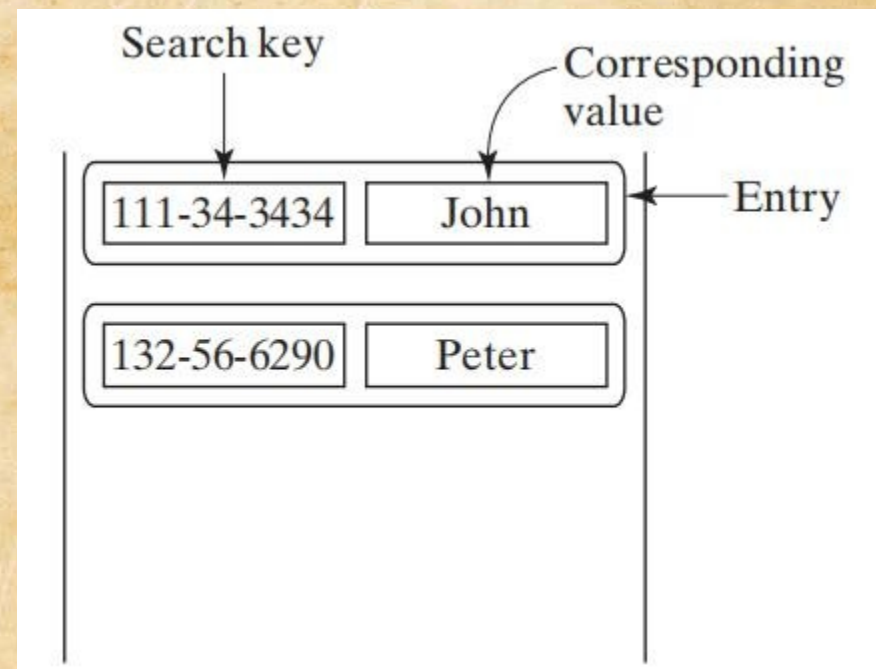


# Estructura i emmagatzematge de Map

## Estructura

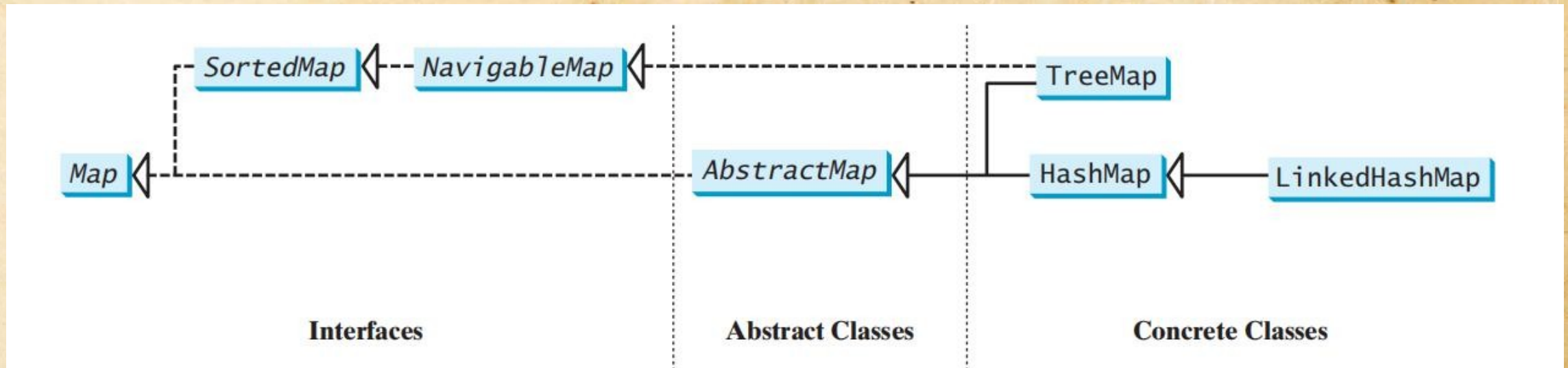
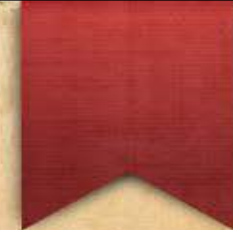


## Emmagatzematge





# Classes concrètes de Map





# Interfície Map

- Retorna true si la clau està en el Map
- Retorna true si el valor està almenys una vegada en el Map
- Retorna un Set compost pels registres (Entry) del Map (clau i valor) en forma d'interfície Map.Entry
- Obté el valor associat a una clau
- Retorna un Set compost per les claus del Map
- Afegeix una nova parella clau/valor (retorna el valor previ, si existeix, associat a la clau)
- Afegeix un grup de parelles clau/valor
- Elimina un element del Map (retorna el valor eliminat)
- Retorna els valors del Map en forma de Collection

«interface»  
*java.util.Map.Entry<K,V>*

+getKey(): K  
+getValue(): V  
+setValue(value: V): void

«interface»  
*java.util.Map<K,V>*

+clear(): void  
+containsKey(key: Object): boolean  
+containsValue(value: Object): boolean  
+entrySet(): Set<Map.Entry<K,V>>  
+get(key: Object): V  
+isEmpty(): boolean  
+keySet(): Set<K>  
+put(key: K, value: V): V  
+putAll(m: Map<? extends K,? extends V>): void  
+remove(key: Object): V  
+size(): int  
+values(): Collection<V>



# Classe HashMap

- No pot contindre claus duplicades.
- Permet valors nuls i una clau nul·la.
- És una col·lecció no ordenada.
- No és segur per a fils.

Veure exemples a:

<https://www.callicoder.com/java-hashmap/>

**java.util.HashMap<K,V>**

+HashMap()

+HashMap(initialCapacity: int,loadFactor: float)

+HashMap(m: Map<? extends K, ? extends V>)



# Classe LinkedHashMap

- Hereta de HashMap
- A diferència de HashMap, l'ordre d'iteració dels elements en un LinkedHashMap és previsible (**ordre d'inserció**).

Veure exemples a:

<https://www.callicoder.com/java-hashmap/>

**java.util.HashMap<K,V>**

+HashMap()  
+HashMap(initialCapacity: int,loadFactor: float)  
+HashMap(m: Map<? extends K, ? extends V>)



**java.util.LinkedHashMap<K,V>**

+LinkedHashMap()  
+LinkedHashMap(m: Map<? extends K,? extends V>)  
+LinkedHashMap(initialCapacity: int,  
loadFactor: float, accessOrder: boolean)



# Classe TreeMap

- Un TreeMap sempre s'ordena en funció de les claus.
- Segueix l'ordre natural de les claus. També es pot proporcionar un “Comparator” personalitzat al TreeMap en el moment de crear-lo.
- Un TreeMap no pot contindre claus duplicades.
- TreeMap no pot contindre cap clau nul·la. Tanmateix, pot tenir valors nuls.

**java.util.TreeMap<K,V>**

```
+TreeMap()  
+TreeMap(m: Map<? extends K,? extends V>)  
+TreeMap(c: Comparator<? super K>)
```

Veure exemples a:

<https://www.callicoder.com/java-treemap/>