
Estruturas de Informação

JAVA Collections Framework

Fátima Rodrigues

mfc@isep.ipp.pt

Departamento de Engenharia Informática (DEI/ISEP)

Java Collections Framework (JCF)

Unified architecture for representing and manipulating collections

A collection is an object that maintains references to others objects

- Essentially a subset of data structures

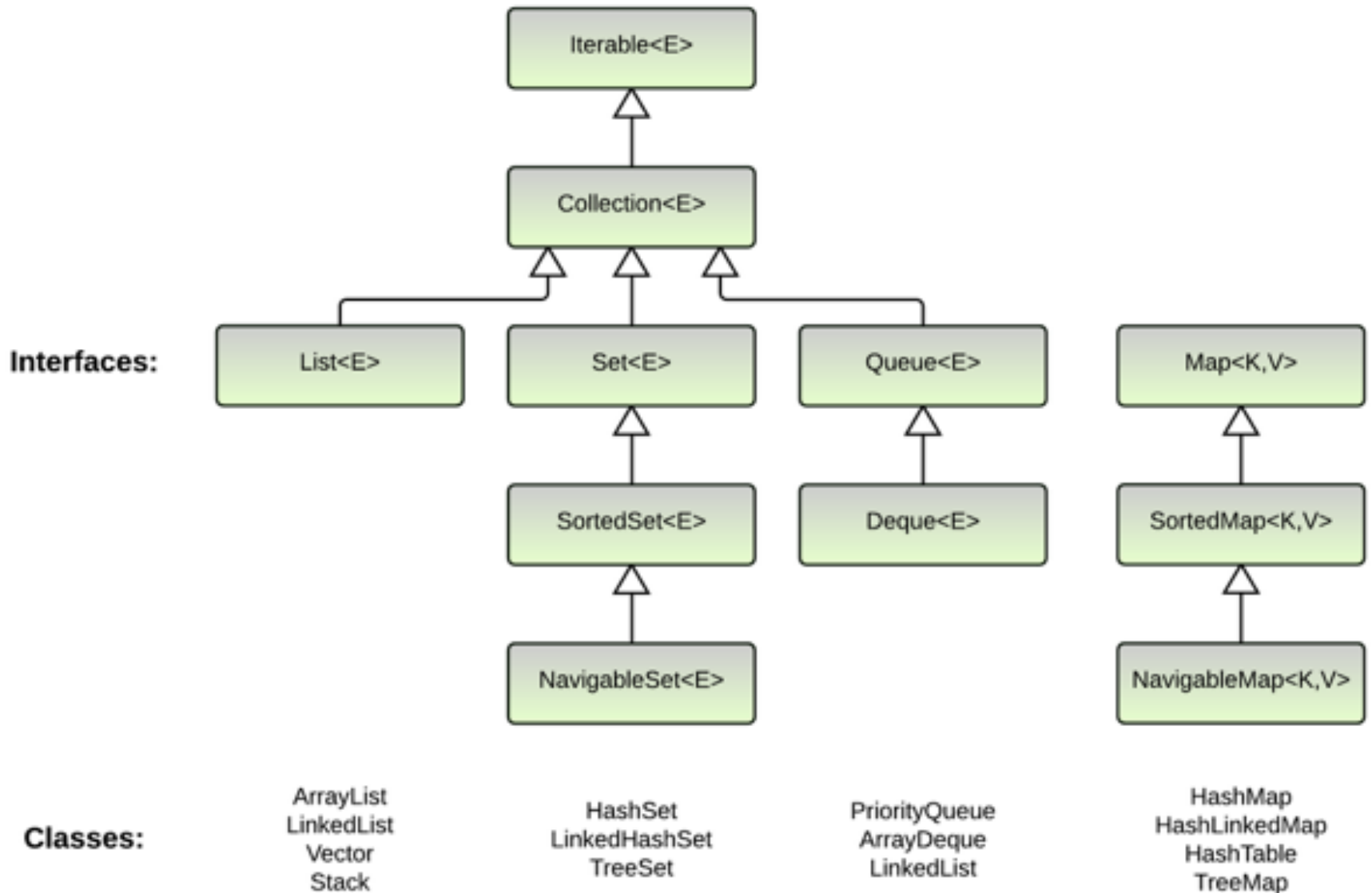
JCF forms part of the **java.util** package and provides:

- interfaces
- abstract classes
- implementations (concrete classes)
- methods for manipulating collections

Offer:

- Standard APIs
- Conversions between structures
- It **reduces the programming effort**, because it offers a lot of code that can be reused

Java Collections Framework



Generics

Generic Class or Parameterized Class

- The interfaces, classes and methods of Java Collections Framework allow *types* to be passed as *parameters*

```
List<Employee> teamwork = new ArrayList<>();
```

- With the type parameter, the ***compiler*** ensures that **we use the collection with objects of a compatible type only**
- Object type errors are now detected at **compile time**, rather than throwing casting exceptions at runtime
- Another benefit is that we won't need to cast the objects we get from the collection:

```
Employee e = teamwork.get(0);
```

Multiple Type Parameters

- A generic class can have any number of type parameters

```
public class Pair<T,S> {  
    private T first;  
    private S second;  
    // Constructors:  
    public Pair() {  
        first = null;  
        second = null; }  
  
    public Pair(T firstElem, S secondElem) {  
        first = firstElem;  
        second = secondElem; }  
    ...  
    public boolean equals(Object otherObj) {  
        if (otherObj == null)  
            return false;  
        if (getClass() != otherObj.getClass())  
            return false;  
  
        Pair<T,S> otherPair = (Pair<T,S>) otherObj;  
        return (first.equals(otherPair.first) &&  
                second.equals(otherPair.second)); }  
}
```

Limitations on type parameter usage

- The type plugged in for a type parameter must always be a **reference type**: it cannot be a primitive type such as `int`, `double`,....
- The type parameter **cannot be used as a constructor** name or like a constructor:

```
T object = new T(); //wrong!
```

```
Pair<String,Integer> filmrating = new Pair<>("Magnolia",8);
```

- Arrays such as the following are illegal:

```
T[] a = new T[10]; //wrong!
```

```
Pair<String,Integer>[] a = new Pair<String,Integer>[10]; //wrong!
```

- Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes

```
ArrayList<Pair<String,Integer>> filmsrating = new ArrayList<>(10);
```

Bounds for Type Parameters

To ensure that only classes that implement the **Comparable** interface are plugged in for **T**, the class must be defined as follows:

```
public class Example <T extends Comparable>
```

- "**extends Comparable**" serves as a *bound* on the type parameter **T**
- Any attempt to plug in a type for **T** which does not implement the **Comparable** interface will result in a **compiler error message**
- A bound on a type may be a class name (rather than an interface name)

```
public class Example <T extends Class1>
```

- A type parameter can have multiple bounds, If one of the bounds is a class, it must be specified first

```
public class Two<T1 extends Class1 & Comparable>
```


Generic Classes and Exception

- It is not permitted to create a generic class with **Exception, Error, Throwable, or any descendent class of Throwable**
- A generic class cannot be created whose objects are throwable

```
public class GEx<T> extends Exception    //wrong!
```

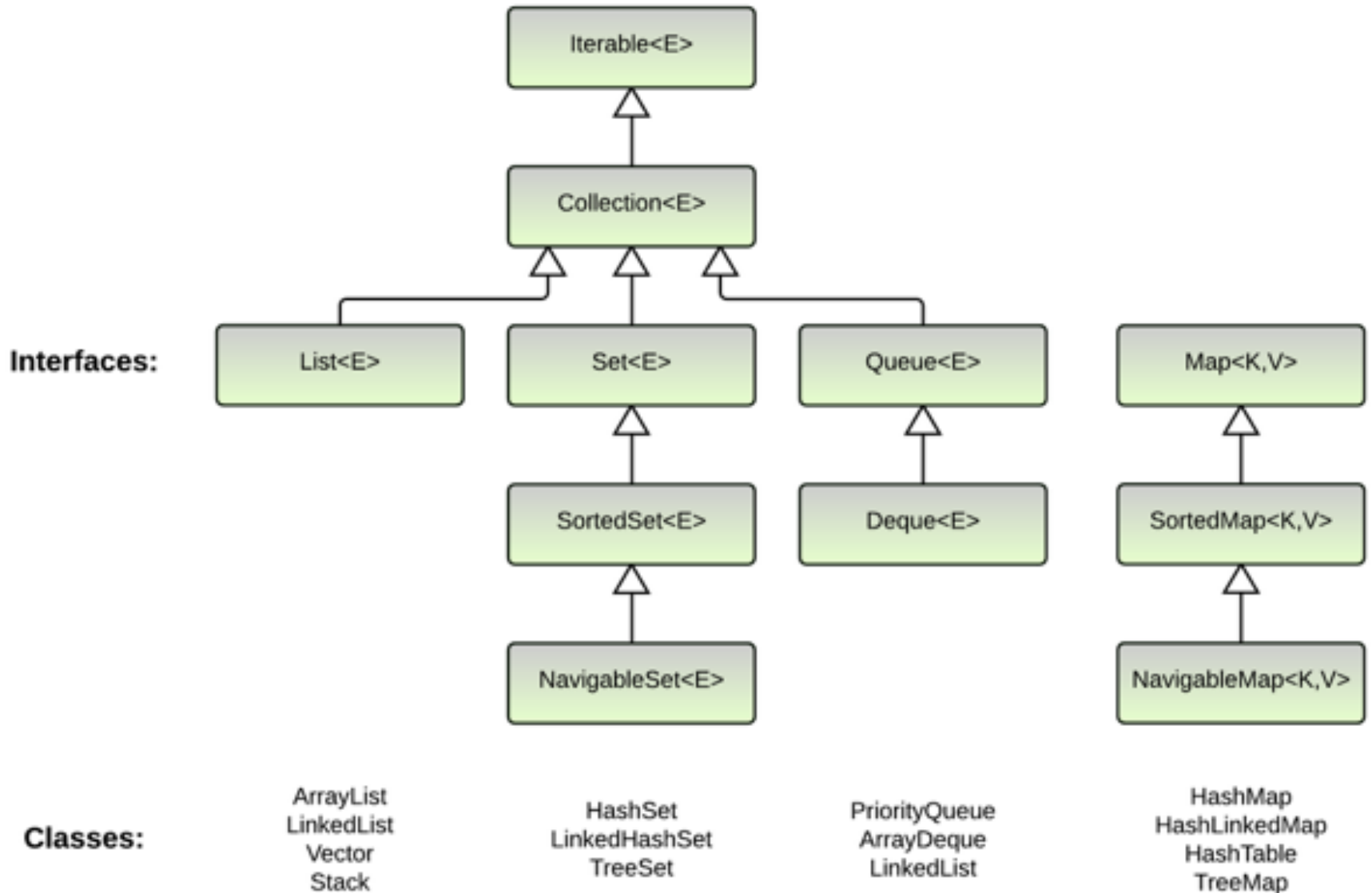
This will generate a compiler error message

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class
- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class
 - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter
 - The **type parameter** of a generic method is **local to that method**, not to the class
- The type parameter must be placed (in angular brackets) after all the modifiers, and before the returned type:

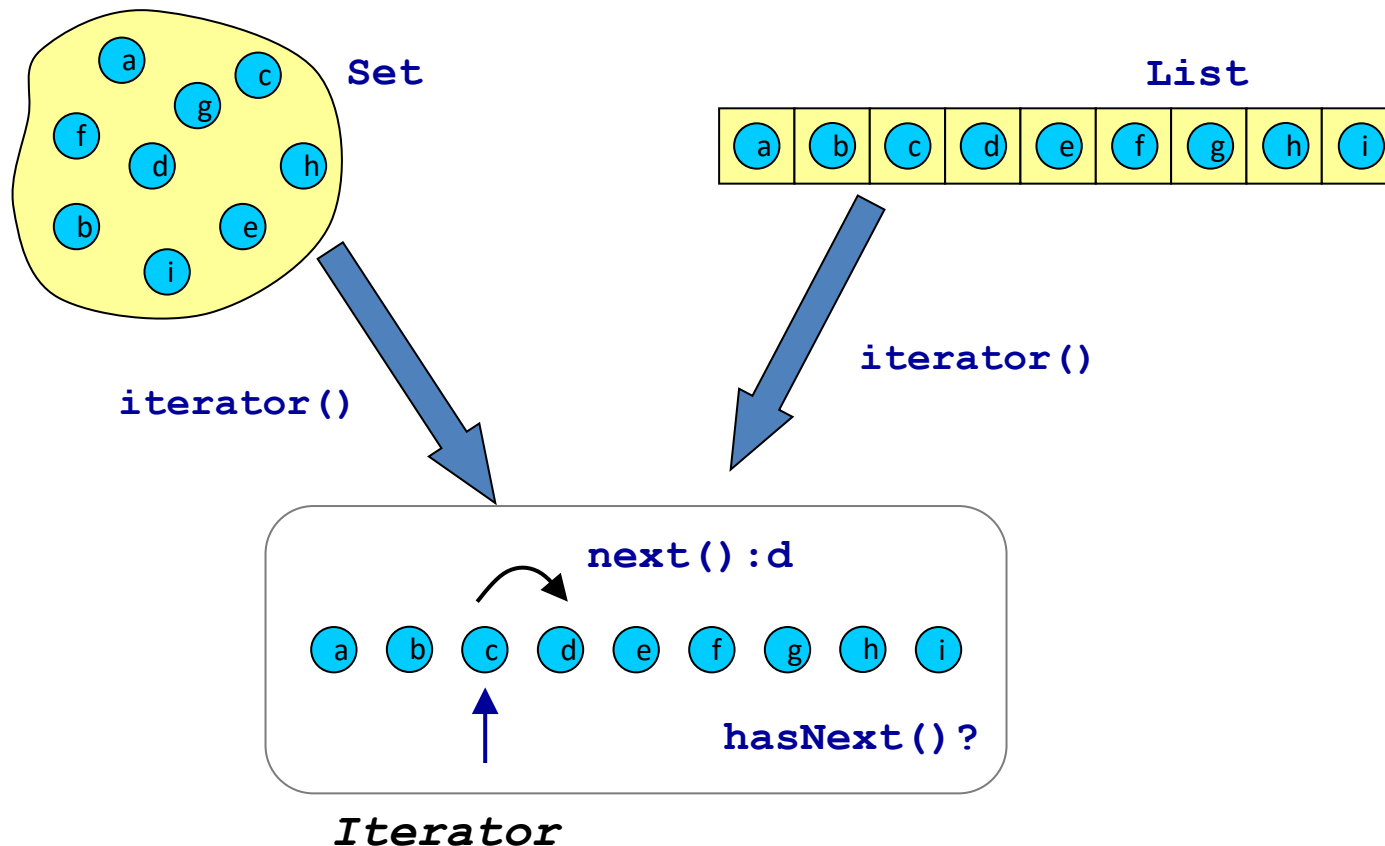
```
public static<T> T genMethod(T[] a)
```

Java Collections Framework



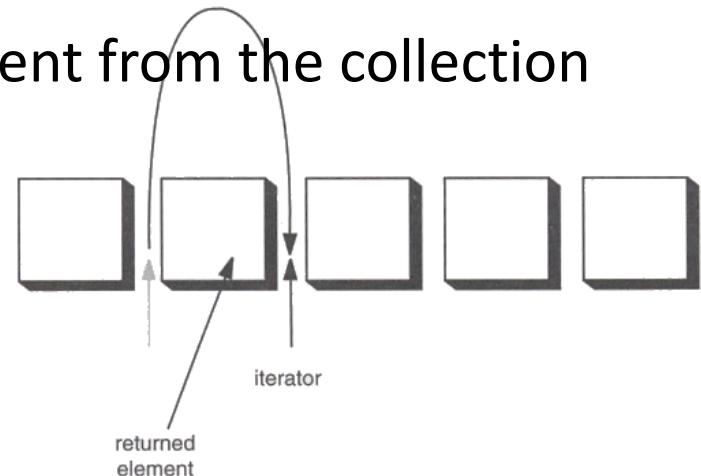
java.util.Iterable<E>

Iterators provide a generic way to traverse through a collection regardless of its implementation



Iterator Interface

- Defines three fundamental methods
 - Object next() - returns the next element
 - boolean hasNext() - returns true if there is a next element, otherwise, returns false
 - void remove() – removes it position
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() “reads” an element from the collection
 - Then you can use it or remove it



Using an Iterator

- Code snippet for collection iteration:

```
public void displayContents(Collection<T> content) {  
    Iterator<T> it = content.iterator();  
    while(it.hasNext()) {  
        T item = it.next();  
        System.out.println(item);  
    }  
}
```

<<interface>> Iterator<E>
+hasNext():boolean +next():E +remove():void

- Above method takes in an object whose class implements Collection
 - List, ArrayList, LinkedList, Set, HashSet, TreeSet, Queue, MyOwnCollection, etc.
- We know any such object can return an Iterator through method iterator()
- We don't know the exact implementation of Iterator we are getting, but **we don't care**, as long as it provides the methods next() and hasNext()

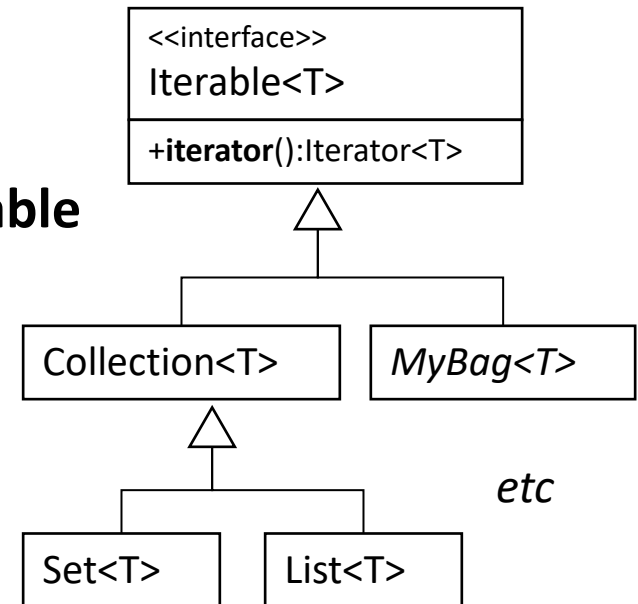
Iterable<T>

```
for (T item : content) {  
    System.out.println(item);  
}
```

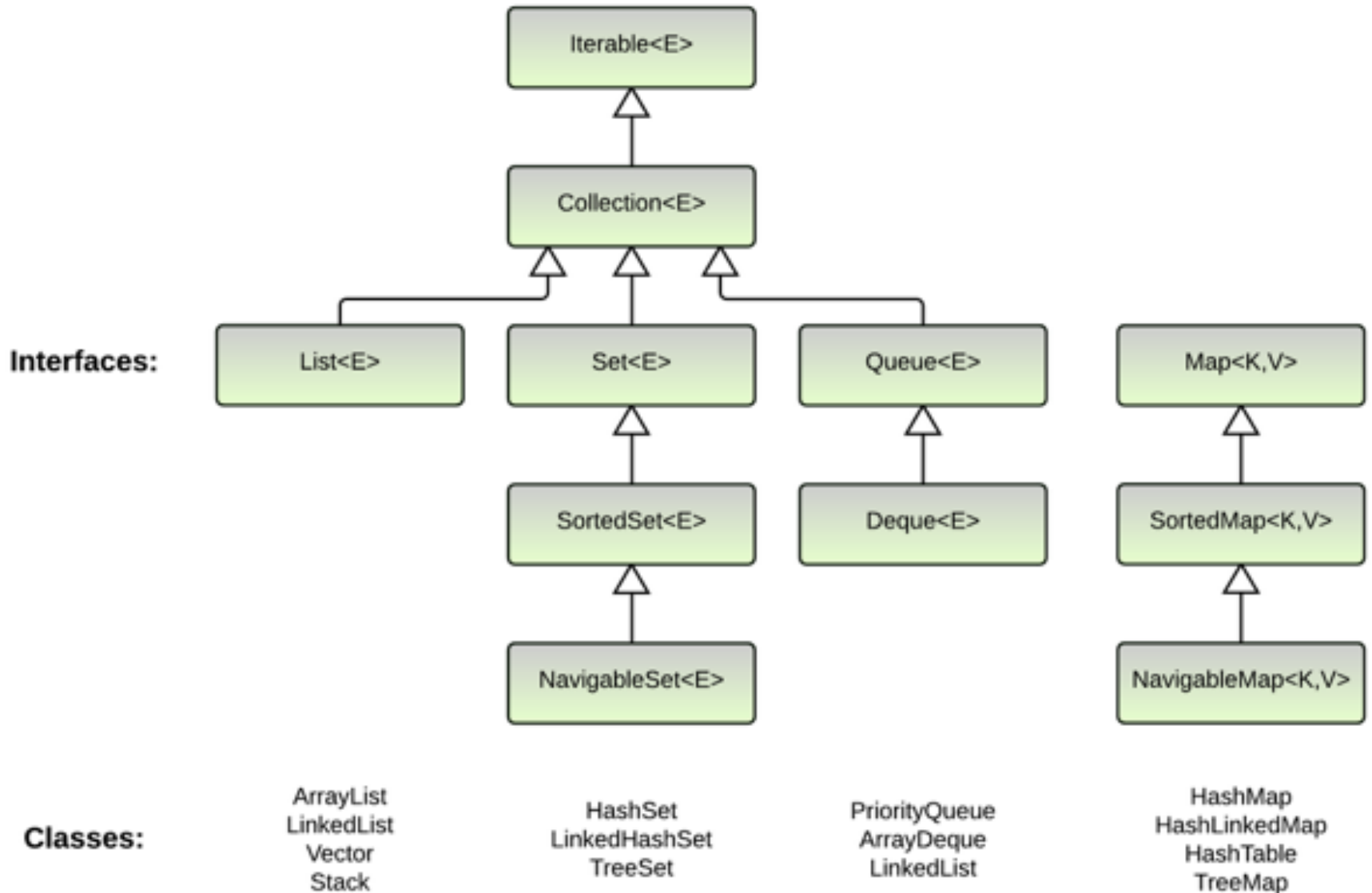
=

```
Iterator<T> it = items.iterator();  
while(it.hasNext()) {  
    Item item = it.next();  
    System.out.println(item);  
}
```

- This is called a “**for-each**” statement
 - For each **item** in **items**
- This is possible as long as items is of type **Iterable**
 - Defines single method **iterator()**
- **Collection** (and hence all its subinterfaces) implements **Iterable**



Java Collections Framework



Collection Interface

- Defines fundamental methods
 - `int size();`
 - `boolean isEmpty();`
 - `boolean contains(Object element);`
 - `boolean add(Object element); // Optional`
 - `boolean remove(Object element); // Optional`
 - `Iterator iterator();`
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

java.util.Collections

- The Collections class offers many very useful utilities and algorithms for manipulating and creating collections
 - **Sorting** lists
 - Index searching
 - Finding min/max
 - Reversing elements of a list
 - Swapping elements of a list
 - Replacing elements in a list
 - Other nifty tricks
- Saves you having to implement them yourself → **reuse**

Sorting

Collections.sort()

What types of objects is it possible to sort?

- Anything that has an **ordering**

Two sort() methods: sort a given List according to either

1. *natural ordering* of elements or
2. an externally defined ordering

1) `public static <T extends Comparable<? super T>> void sort(List<T> list)`

2) `public static <T> void sort(List<T> list, Comparator<? super T> c)`

1. Only accepts a List parameterised with type implementing **Comparable**
2. Accepts a List parameterised with any type as long as you also give it a **Comparator** implementation that defines the ordering for that type

java.lang.Comparable<T>

- A **generic interface** with a single method: **int compareTo(T)**
 - Return 0 if this == other
 - Return **any positive integer** if this > other
 - Return **any negative integer** if this < other
- Implement this interface to define **natural ordering** on objects of type T

```
public class Pair <T extends Comparable<T>, S extends Comparable<S>>
    implements Comparable< Pair<T,S> >{
    ...
    public int compareTo(Pair<T,S> other) {
        return this.getSecond().compareTo(other.getSecond());
    }
}
```

java.lang.Comparable<T>

```
Pair<String,Integer> film1 = new Pair<>("Magnolia",8);  
Pair<String,Integer> film2 = new Pair<>("Crocodile Dundee",3);  
Pair<String,Integer> film3 = new Pair<>("Paris Texas",6);
```

```
ArrayList<Pair<String,Integer>> filmsrating = new  
ArrayList<>(10);
```


```
filmsrating.add(film1);  
filmsrating.add(film2);  
filmsrating.add(film3);  
filmsrating.sort(filmsrating);
```

Pair: (Crocodile Dundee, rate 3)
Pair: (Paris Texas, rate 6)
Pair: (Magnolia, rate 8)

```
System.out.println("\nFilms sorted by rating");  
for (Pair<String,Integer> p : filmsrating )  
    System.out.println("Pair: (" + p.getFirst() + ", rate " +  
                        p.getSecond() + ")");
```

```
List<CD> albums = new ArrayList<CD>();  
albums.add(new CD("A Head Full of Dreams","Coldplay",2.80));  
//etc...
```

```
Collections.sort(albums);
```

←  CD does not implement a Comparable interface

java.util.Comparator<T>

- Useful if the type of elements to be sorted is not Comparable, or we want to **define an alternative ordering**
- Also a generic interface that defines methods **compare(T,T)** and **equals(Object)**
 - Usually only need to define **compare(T,T)**

<<interface>> Comparator<T>
+ compare (T o1, T o2):int + equals (Object other):boolean

```
public static class PairComparator<T extends Comparable<T>, S  
extends Comparable<S>> implements Comparator < Pair<T,S> > {  
  
    public int compare(Pair<T,S> p1, Pair<T,S> p2) {  
        return p1.getFirst().compareTo(p2.getFirst());  
    }  
}
```

Comparator defined using
Comparable ☺

Comparator sorting

implements `Comparator<CD>`

```
Collections.sort(filmsrating, new PairComparator());  
System.out.println("\nFilms sorted by Title");  
for (Pair<String,Integer> p : filmsrating )  
    System.out.println("Pair: (" + p.getFirst() + ", rate " +  
                        p.getSecond()+")");
```

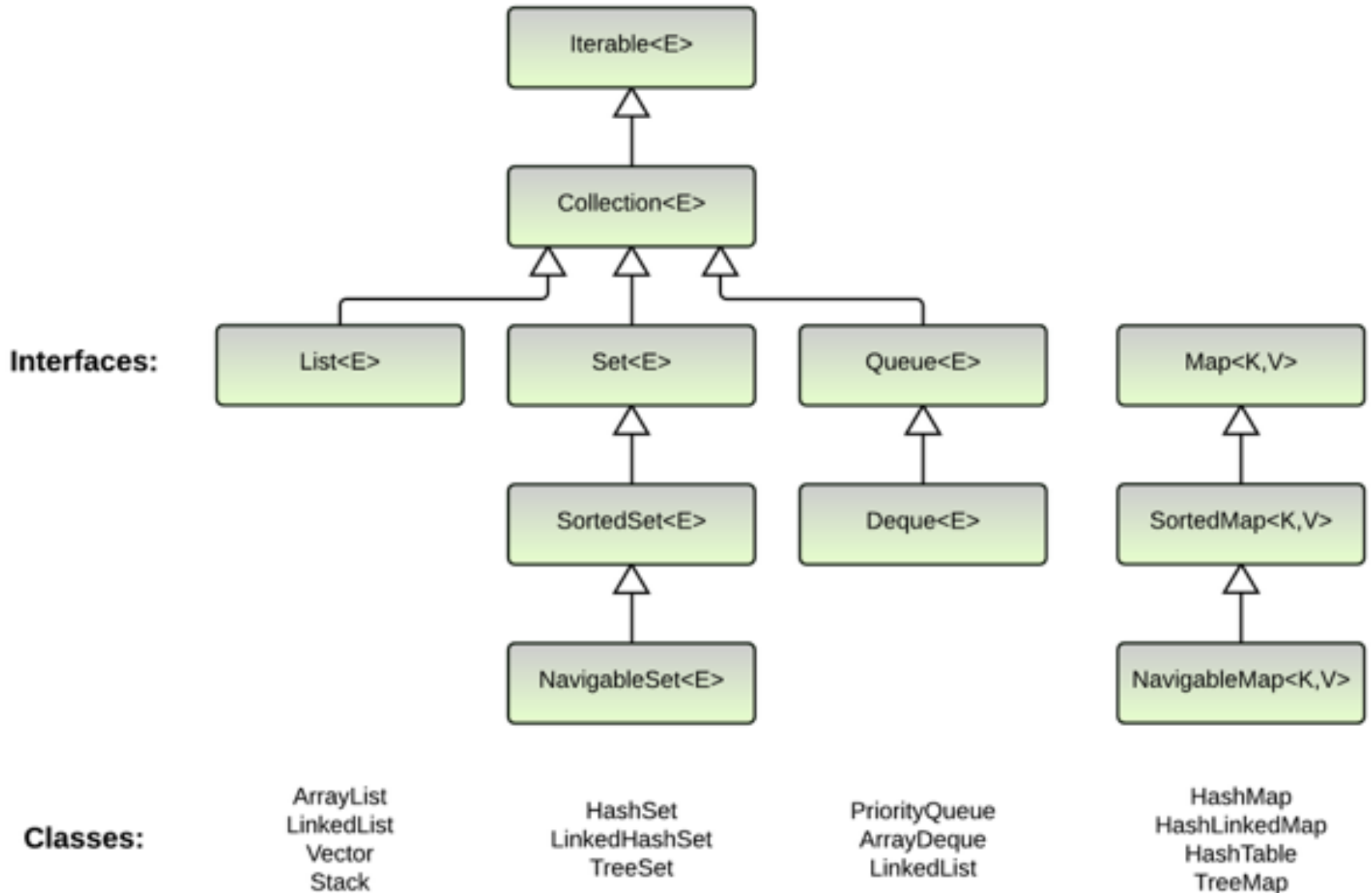
Pair: (Crocodile Dundee, rate 3)
Pair: (Magnolia, rate 8)
Pair: (Paris Texas, rate 6)

- Note, in `sort()`, `Comparator` overrides natural ordering
 - i.e. even if we define natural ordering for `Pair`, the given comparator is still going to be used instead
 - (On the other hand, if you give **null** as `Comparator`, then natural ordering is used)

Differences between comparable and comparator

	Comparable	Comparator
Sort sequence	Only one sort sequence	Many sort sequences
Methods	<code>compareTo(Object o)</code>	<code>Compare(Object o1, Object o2)</code>
Objects needed For comparison	Compares “this” reference with the object provided	Compares two objects
Modify classes	Modify the class whose instances are to sort	Build a class separate from the class whose instances are to sort
Package	Java.lang Should be provided by default	Java.util Should be used as an utility to sort objects

Java Collections Framework



List<E> implementations

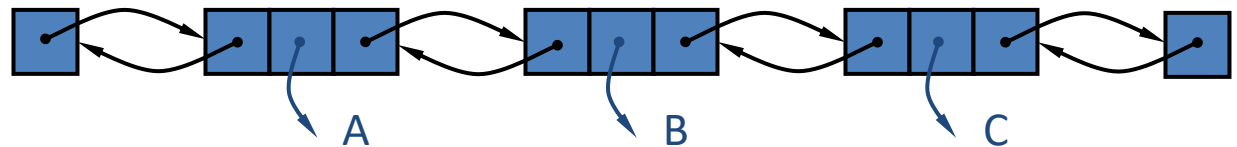
Two concrete implementations of the List ADT in the Java Collection:

- **java.util.ArrayList**



An array is fixed once it is created

- **java.util.LinkedList**



A list can grow or shrink dynamically

List Interface

- The List interface **adds the notion of order** to a collection
- The user of a list has control over where an element is added in the collection
- With a list it is possible:
 - to store **duplicate** elements
 - to specify where the element is stored
 - to access the element by index
- Provides a ListIterator to step through the elements in the list

```
<<interface>>
```

```
List<E>
```

```
+add(E):boolean
```

```
+remove(Object):boolean
```

```
+get(int):E
```

```
+indexOf(Object):int
```

```
+contains(Object):boolean
```

```
+size():int
```

```
+iterator():Iterator<E>
```

```
etc...
```

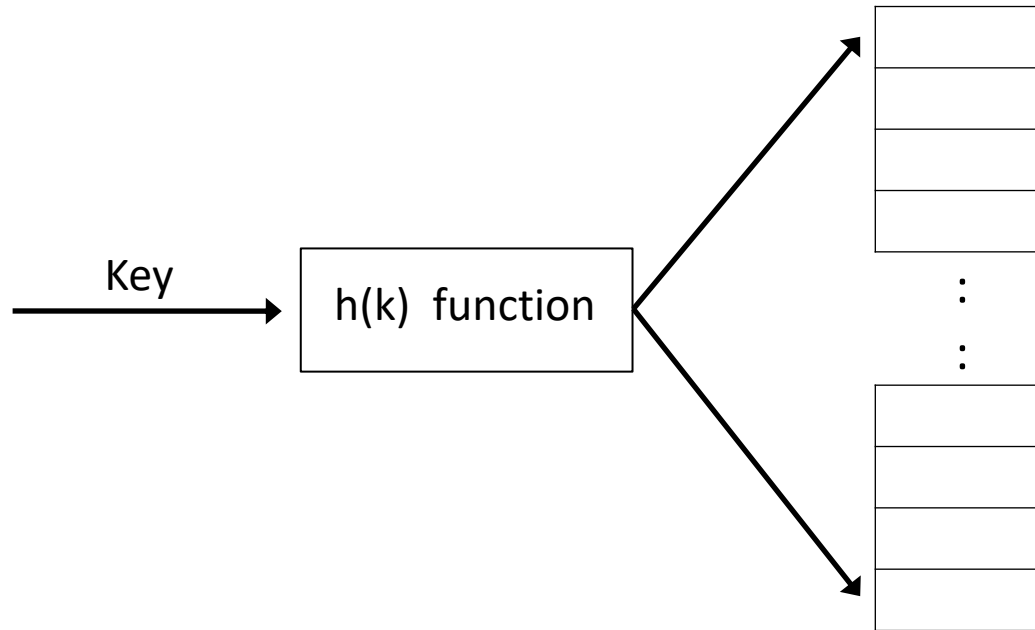
ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
 - void **add(Object o)** - Inserts object into the list in front of it position
 - boolean **hasPrevious()**
 - Object **previous()**
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

Hash Table

Hash Table

- Is a data structure where the location of an item is determined
 - Directly as a function of the item itself
- If the hash table is implemented as an array A of size N the **hash function $h(k)$** must map a key k into an integer range $0, 1, \dots, N-1$



From Keys to Indices

- The mapping of keys to indices of a hash table is called a **hash function**
- An essential requirement of a hash function is to map equal keys to equal indices
- A “good” hash function must be:
 - easy and fast to compute
 - minimize the **probability of collisions**, by distribute the items evenly throughout the hash table
- A perfect hash function can be constructed if we know in advance all the keys to be stored in the table (almost never...)

Hash Functions: Examples

A hash function is a composition of two functions:

1. Hash code:

- Key = Character: char value cast to an int -> it's ASCII value
- Key = Date: value associated with the current time
- Key = Double: value generated by its bitwise representation
- Key = Integer: the int value itself
- Key = String: a folded sum of the character values
- Key = URL: the hash code of the host name

1. Compression function

Maps the hash code to a valid Index for example, modulus operator (%) with table size

```
idx = hash(val) % size;
```

To get a good distribution of indices, **table size** should be a **prime number**

Collision

- A collision occurs when two distinct items are mapped to the same position
- Example: store six elements in a **eight** element array, where the hash function converts the 3rd letter of each name to an index

Alfred	$f = 5 \% 8 = 5$
Alessia	$e = 4 \% 8 = 4$
Amina	$i = 0 \% 8 = 0$
Andy	$d = 3 \% 8 = 3$
Aspen	$p = 7 \% 8 = 7$
Aimee	$m = 4 \% 8 = 4$

Collisions Resolution

There are two general approaches to resolving collisions:

1. Open address hashing:

If that position is filled, next position is examined, then next, and so on until an empty position is filled

Amina			Andy	Alessia	Alfred		Aspen
0	1	2	3	4	5	6	7
aiqy	bjrz	cks	dlt	emu	fnv	gpw	hpq

To add: Aimee

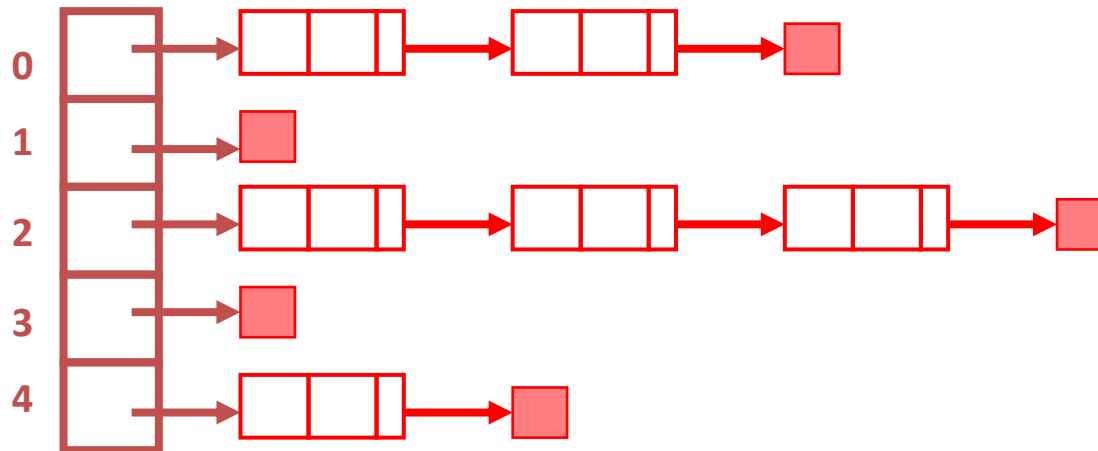
Hashes to							
				Placed here			
Amina			Andy	Alessia	Alfred	Aimee	Aspen
0	1	2	3	4	5	6	7
aiqy	bjrz	cks	dlt	emu	fnv	gpw	hpq

Collisions Resolution

There are two general approaches to resolving collisions:

2. Chaining (or buckets): keep a collection at each table entry

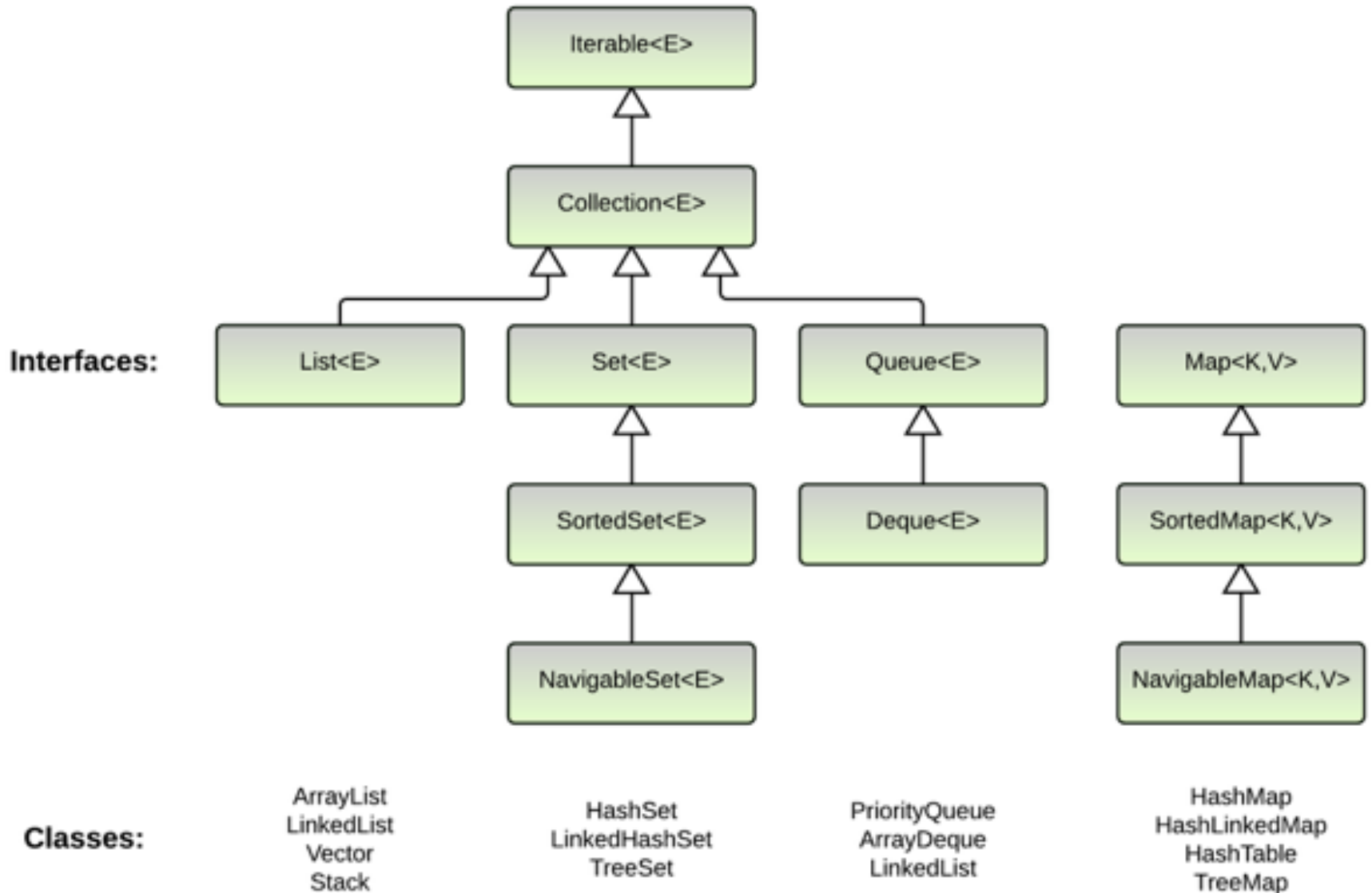
Each position is viewed as a container of a list of items, not a single item. All items in this list share the same hash value



Java Hash function

- Java provides a suitable hash function `hashCode()` defined in **Object class** and inherited by all subclasses, which typically returns the **32-bit memory address of the object**
- If a class **overrides the equals method** defined in class Object it is also necessary to **override the hashCode** method to make **HashSet and HashMap** work correctly
- The `hashCode()` method **should be suitably redefined by classes**

Java Collections Framework

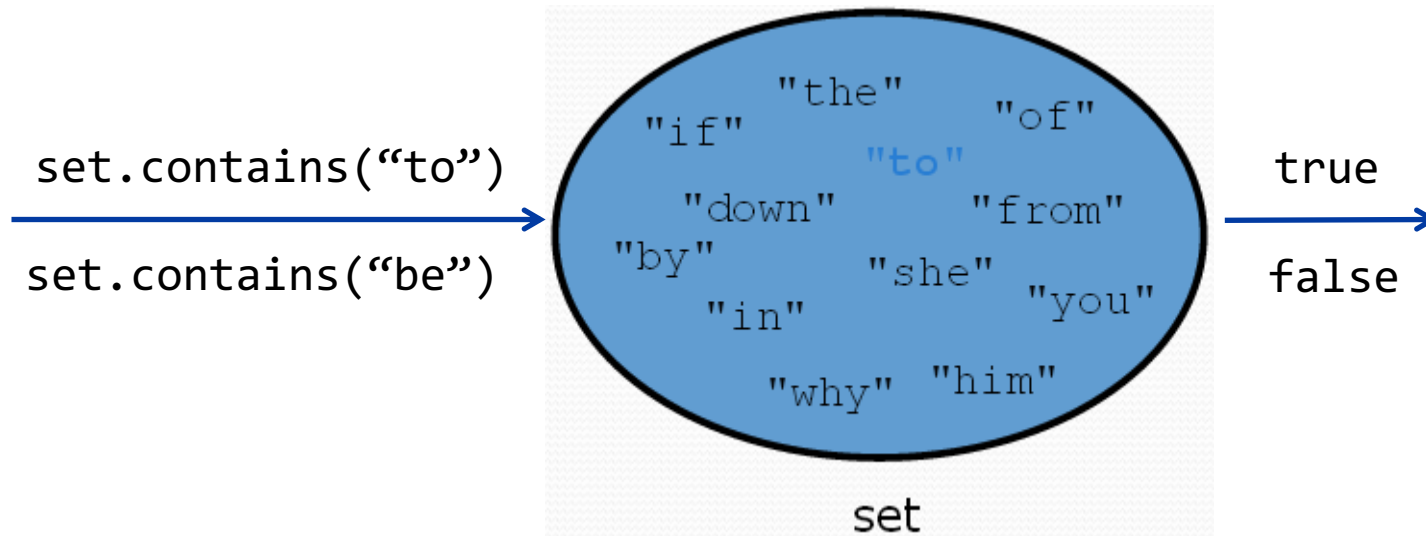


Set

Set is a collection of unique values (no duplicates allowed) that can perform the following operations efficiently:

- add, remove, search (contains)

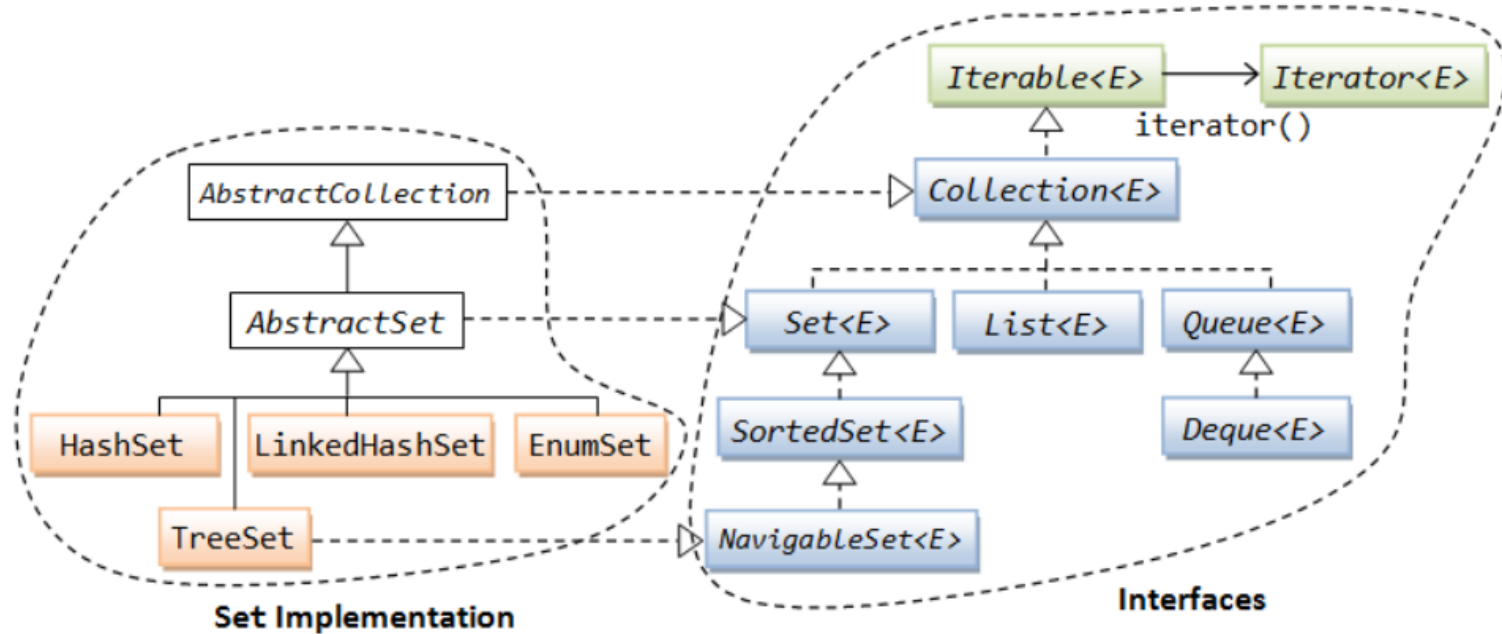
We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



Provides an Iterator to step through the elements in the Set

- No guaranteed order in the basic Set interface
- There is a SortedSet interface that extends Set

Set<E> Interfaces



The Set<E> interface abstract methods:

```
boolean add(E o) //add the specified element if it is not already present
boolean remove(Object o) // remove the specified element if it is present
boolean contains(Object o) // return true if it contains o

// Set operations
boolean addAll(Collection<? extends E> c) //Set union
boolean retainAll(Collection<?> c) //Set intersection
boolean removeAll(Collection<?> c) //Set difference
```


Set<E> Implementations

- In Java, sets are represented by Set type in java.util

Set is implemented by:

- **HashSet:** implemented using a "hash table" an array of linked lists
 - elements are stored in unpredictable order
- **TreeSet:** implemented using a "red black tree"
 - elements are stored in sorted order
- **LinkedHashSet:** stores the elements in a linked-list hash table
 - stores in order of insertion

Set<E> Implementations

```
Set<String> s1 = new HashSet<>();  
s1.add("DD"); s1.add("EE"); s1.add("BB"); s1.add("CC");  
System.out.println("set s1: " + s1);
```

set s1: [DD, EE, BB, CC]

```
Set<String> s2 = new TreeSet<>();  
s2.add("DD"); s2.add("FF"); s2.add("AA"); s2.add("KK");  
s2.add("FF");  
System.out.println("set s2: " + s2);
```

set s2: [AA, DD, FF, KK]

```
if (s2.retainAll(s1))  
    System.out.println("Intersection S1 S2: " + s2);
```

Intersection S1 S2: [DD]

TreeSet<E> (SortedSet<E>)

- TreeSet guarantees that all elements are ordered (sorted) at all times
 - **add()** and **remove()** preserve this condition
 - **iterator()** always returns the elements in a specified order
- Two ways of specifying ordering
 - Ensuring elements have natural ordering (**Comparable**)
 - Giving a **Comparator<E>** to the constructor
- **Caution:** TreeSet considers x and y are duplicates if:
x.compareTo(y) == 0 (or compare(x,y) == 0)

TreeSet construction

```
Set<String> words = new TreeSet<String>();  
words.add("Bats");  
words.add("Ants");  
words.add("Crabs");  
for (String word : words) {  
    System.out.println(word);  
}
```

String has a **natural ordering**,
so empty constructor

But CD doesn't, so you must pass in a Comparator to the constructor

```
Set<CD> albums = new TreeSet<CD>(new PriceComparator());  
albums.add(new CD("Songs of Innocence","U2",new Money(3,50)));  
albums.add(new CD("Overexposed","Maroon 5",new Money(2,80)));  
albums.add(new CD("Space Cowboy","Jamiroquai",new Money(5,00)));  
albums.add(new CD("Maiden Voyage","Herbie Hancock",new Money(4,00)));  
albums.add(new CD("Here's the Deal","Liquid Soul",new Money(2,80)));  
System.out.println("N. CDs "+albums.size());  
for (CD album : albums) {  
    System.out.println(album);  
}
```

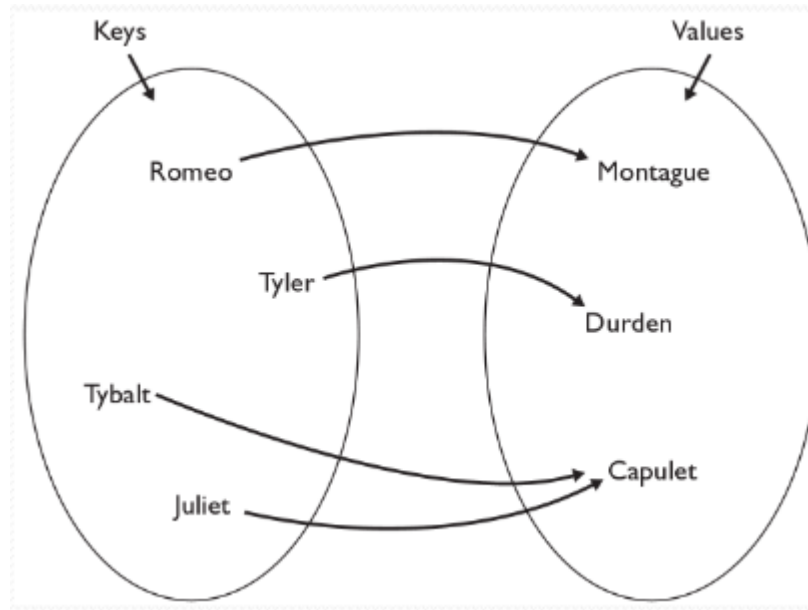
What's the output?

N. CDs 4

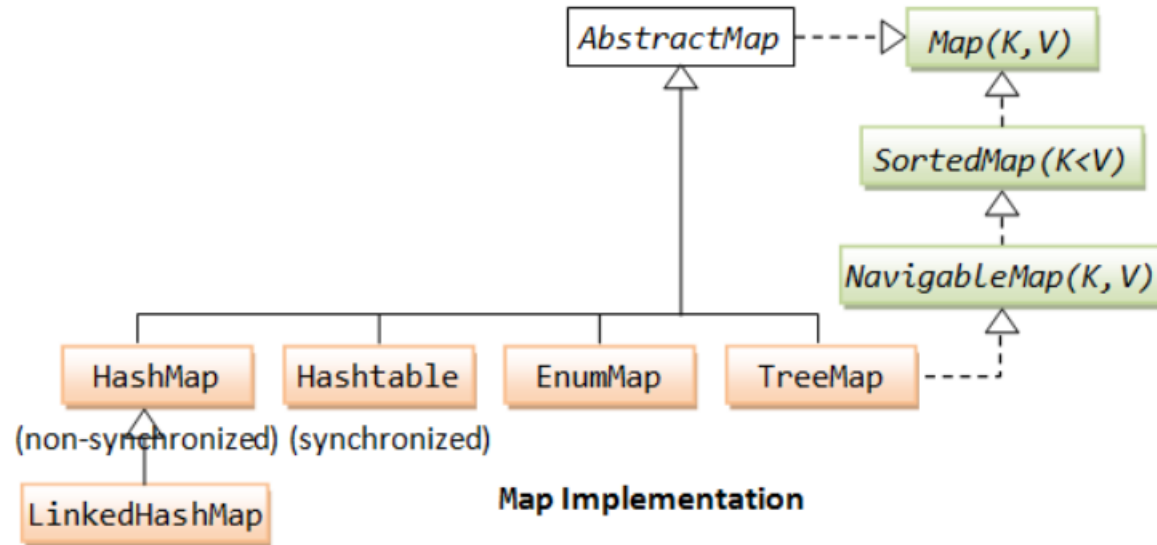
Maroon 5 (2.8); U2 (3.5); Herbie
Hancock (4.0); Jamiroquai (5.0)

Map

- A **map** is a collection of key-value pairs. Each key maps to one and only value. Duplicate keys are not allowed, but duplicate values are allowed
- Maps are similar to linear arrays, except that an array uses an integer key to index and access its elements; whereas a map uses any arbitrary key (such as Strings or any objects)



Map<K,V> Interfaces



Map<K,V> interface abstract methods:

```
V get(Object key)           //Returns the value of the specified key
put(K key, V value)         //Associates the specified value with the specified key
boolean containsKey(Object key)
boolean containsValue(Object value)

// Views
Set<K> keySet()              // Returns a set view of the keys Collection<V>
values()                    // Returns a collection view of the values Set
entrySet()                  // Returns a set view of the key-value
```

Map<K,V> Implementations

in Java, maps are represented by Map type in java.util

Map is implemented by:

- **HashMap:** implemented using an array called a "hash table"
 - no guarantees about the iteration order. It can (and will) even change completely when new elements are added
- **TreeMap:** implemented as a linked "binary tree" structure
 - will iterate according to the "natural ordering" of the key, ie. according to their compareTo() method (or an externally supplied Comparator)
- **LinkedHashMap**
 - will iterate in the order in which the entries were put into the map

HashMap<K, V>

- keys are hashed using **Object.hashCode()**
 - i.e. no guaranteed ordering of keys
- **keySet()** returns a **HashSet**
- **values()** returns an unknown Collection

```
Map<String, Integer> directory = new HashMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
    System.out.print(key+"'s number: ");
    System.out.println(directory.get(key));
}
System.out.println(directory.values());
```


TreeMap<K, V>

- Guaranteed ordering of keys (like TreeSet)
 - In fact, TreeSet is implemented using TreeMap
 - Hence **keySet()** returns a **TreeSet**
- **values()** returns an unknown Collection – ordering depends on ordering of **keys**

```
Map<String, Integer> directory = new TreeMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
    System.out.print(key+"'s #: ");
    System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

Empty constructor
→ natural ordering

4

Loop output?

Bob's #: 1000000
Dad's #: 9998888
Edward's #: 5553535
Mum's #: 9998888

[1000000, 9998888, 5553535, 9998888]

TreeMap with Comparator

As with TreeSet, another way of constructing TreeMap is to give a Comparator necessary for non-Comparable keys

```
Map<CD, Double> ratings
    = new TreeMap<>(new PriceComparator());
ratings.put(new CD("Street Signs", "O", new Money(3, 50)), 8.5);
ratings.put(new CD("Jazzinho", "J", new Money(2, 80)), 8.0);
ratings.put(new CD("Space Cowboy", "J", new Money(5, 00)), 9.0);
ratings.put(new CD("Maiden Voyage", "H", new Money(4, 00)), 9.5);
ratings.put(new CD("Here's the Deal", "LS", new Money(2, 80)), 9.0);

System.out.println(ratings.size());
for (CD key : ratings.keySet()) {
    System.out.print("Rating for "+key+": ");
    System.out.println(ratings.get(key));
}
System.out.println("Ratings: "+ratings.values());
```

4

Ordered by key's price

Depends on key ordering

Double-ended queue or *Deque*

A **queue** is a collection whose elements are added and removed in a specific order, typically in a **first-in-first-out (FIFO)** manner

A **deque** is a double-ended queue that elements can be inserted and removed at both ends (head and tail) of the queue

A Deque can be used:

- as **FIFO queue** via methods:
 - `add(e)/offer(e), remove()/poll(), element()/peek()`
- as **LIFO queue** via methods:
 - `push(e), pop(), peek()`

Queue and Deque implementations

- **PriorityQueue<E>:**

A queue implemented with a **heap** where the elements are ordered based on an ordering specified, instead of FIFO

- **ArrayDeque<E>:**

A queue and deque implemented based on a circular array

- **LinkedList<E>:**

The LinkedList<E> also implements the Queue<E> and Deque<E> interfaces, in addition to the List<E> interface, providing a queue or deque that is implemented as a double-linked list data structure