Programming in Java – Collections

Paolo Vercesi

Technical Program Manager

Agenda

Collections

Lists

Sets

Queues

Maps

Collections

Collections Overview

The collection framework standardize the way in which group of objects are handled

The framework is high-performance, and it must be used by every Java programmer, no need to reinvent the wheel here

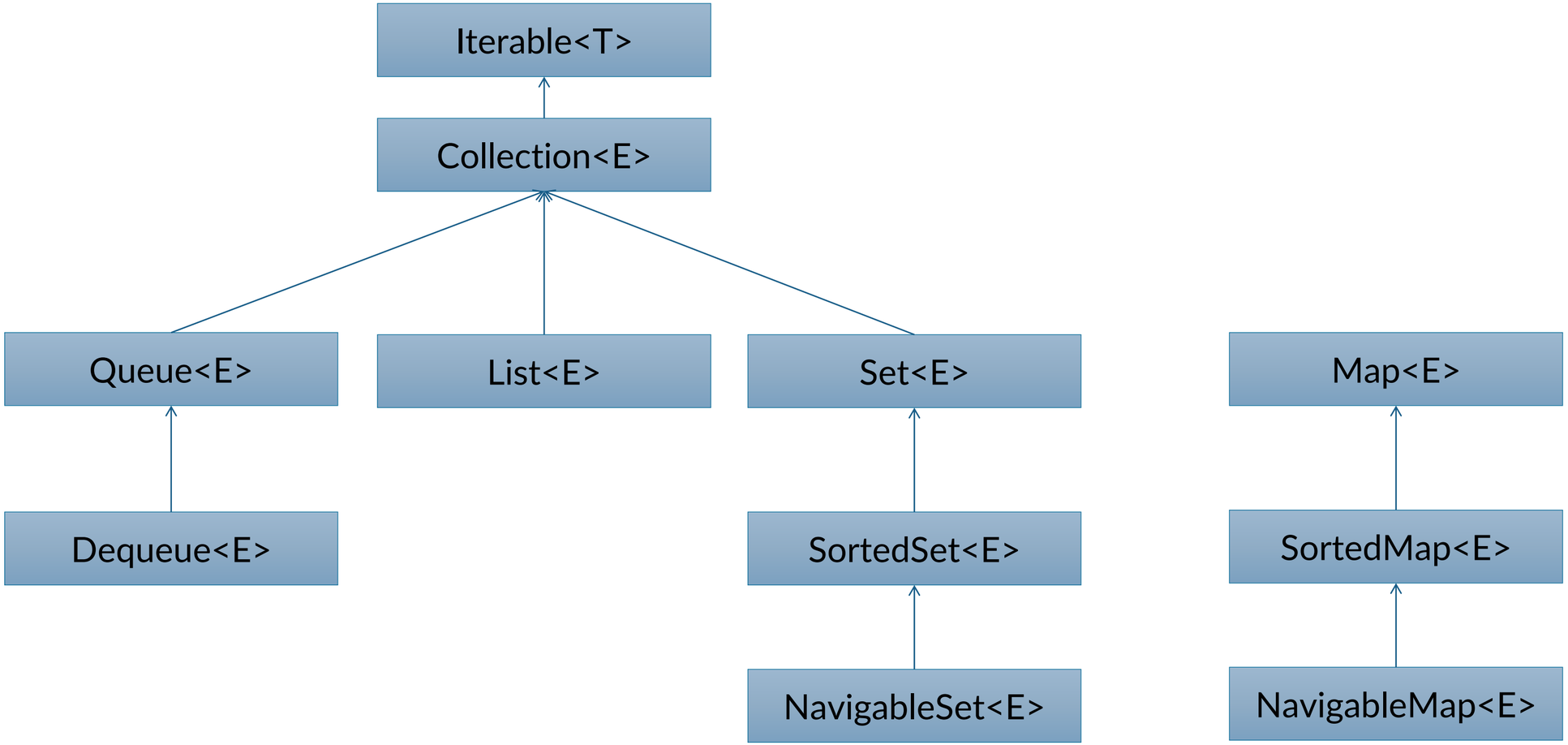
The API is consistent through the different classes and it provides a high degree of interoperability

Maps are parts of the collections framework, even if they are not collections in a strict sense

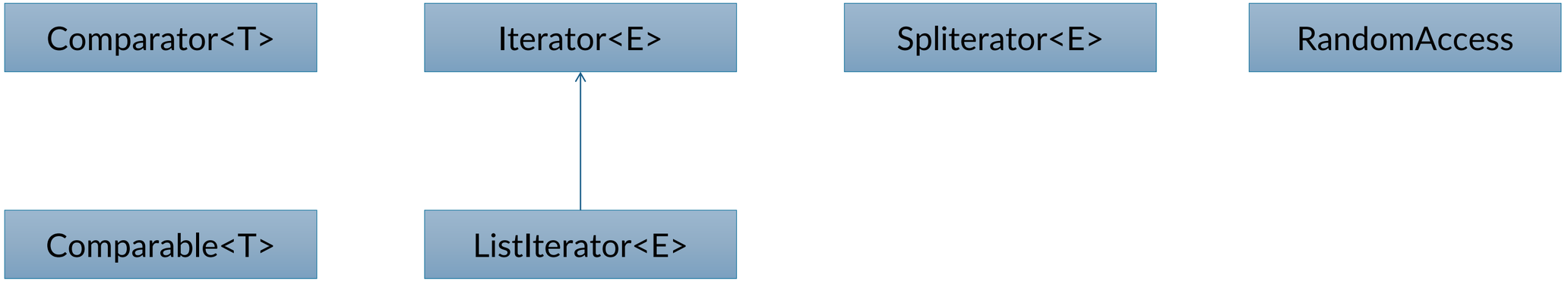
All classes and interfaces in the collection framework are generic

All the classes resides in the `java.util` package

The Collection hierarchy



Other interfaces



The Collection Interface

The Collection interface declares the core methods that all collections have

Querying a collection

```
boolean isEmpty()
int size()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
Iterator<E> iterator()
default Spliterator<E> spliterator()
default Stream<E> stream()
default Stream<E> parallelStream()
Object[] toArray()
default <T> T[] toArray(IntFunction<T[]> generator)
<T> T[] toArray(T[] a)
```

Modifying a collection

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void clear()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
default boolean removeIf(Predicate<? super E> filter)
```

All these methods are optional, and they throw UnsupportedOperationException if the collection is unmodifiable.

The Iterable interface

Iterable objects can be used in for-each loops

```
Iterable<Integer> iterable = List.of(1, 2, 3, 4, 5);

for (Integer integer : iterable) {
    System.out.println(integer);
}
```

The Comparable interface

```
public interface Comparable<T> {

    public int compareTo(T o);
}
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object

The implementor must ensure signum(x.compareTo(y)) == -signum(y.compareTo(x)) for all x and y

The implementor must also ensure that the relation is transitive:
(x.compareTo(y) > 0 && y.compareTo(z) > 0) implies x.compareTo(z) > 0

Finally, the implementor must ensure that
x.compareTo(y)==0 implies that signum(x.compareTo(z)) == signum(y.compareTo(z)), for all z

The Comparator interface

```
public interface Comparator<T> {

    int compare(T o1, T o2);
}
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

How to remember? If they are integer, it is the result of `o1-o2`.

The implementor must ensure that signum(compare(x, y)) == -signum(compare(y, x)) for all x and y.

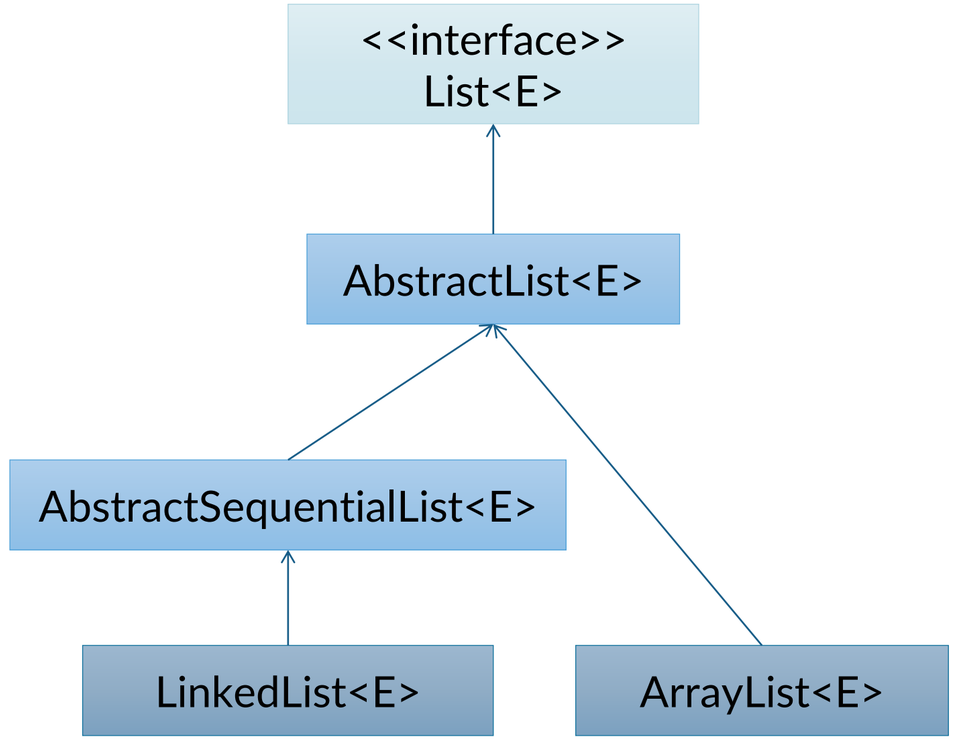
The implementor must also ensure that the relation is transitive:
((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

Finally, the implementor must ensure that
compare(x, y)==0 implies that signum(compare(x, z))==signum(compare(y, z)) for all z.

Lists

Lists

```
public interface List<E> extends Collection<E> {
    …
}
```



The List interface

The List interface extends the Collection interface with list specific methods

```
E get(int index)
E set(int index, E element)
E remove(int index)
boolean addAll(int index, Collection<? extends E> c)
default void replaceAll(UnaryOperator<E> operator)
default void sort(Comparator<? super E> c)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
List<E> subList(int fromIndex, int toIndex) (returns a view)
```

Creating unmodifiable lists

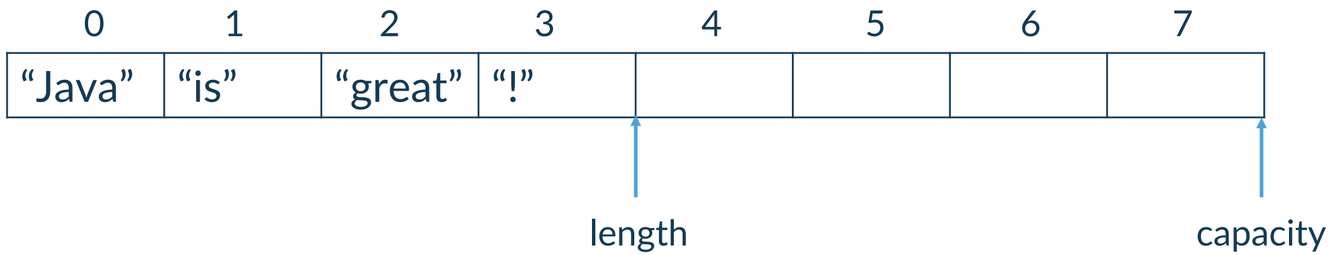
The List interface provides methods to create unmodifiable lists

```
static <E> List<E> copyOf(Collection<? extends E> coll)
static <E> List<E> of()
static <E> List<E> of(E e1)
…
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
```

The ArrayList class

Implements List and RandomAccess

Internally based on arrays

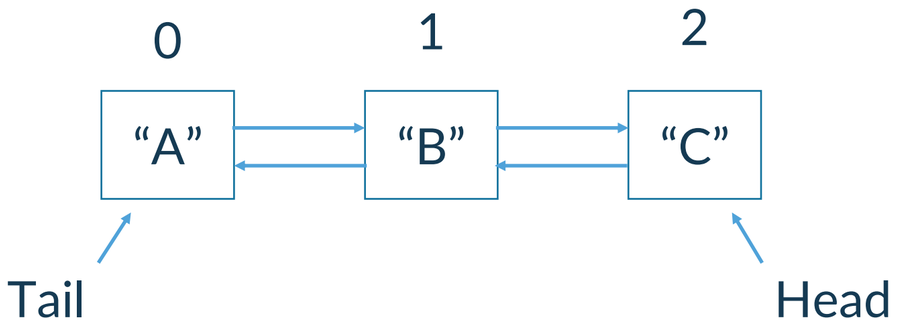



```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
void ensureCapacity(int cap)
void trimToSize( )
```

The LinkedList class

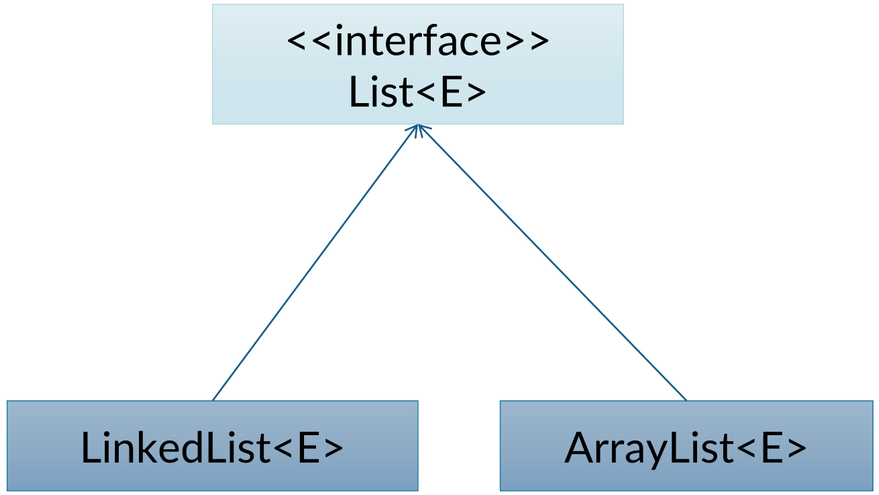
Implements List, Queue and Deque

Double linked list




```
LinkedList( )
LinkedList(Collection<? extends E> c)
```

Lists in summary

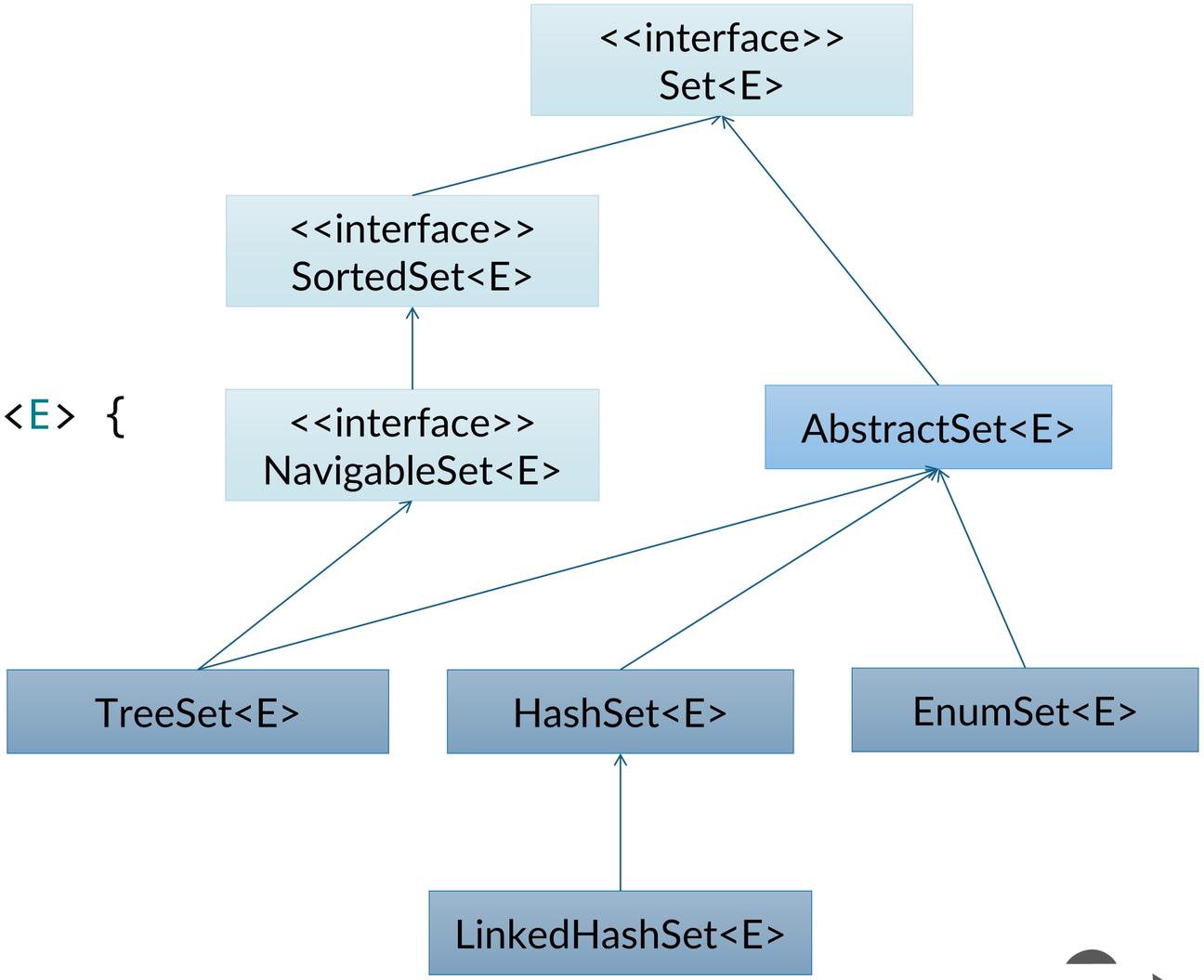


Sets

Sets

The set interface specifies the behavior of a collection that does not allow duplicate elements. The Set interface extends Collection and it does not specify any additional instance method of its own.

```
public interface Set<E> extends Collection<E> {
    …
}
```



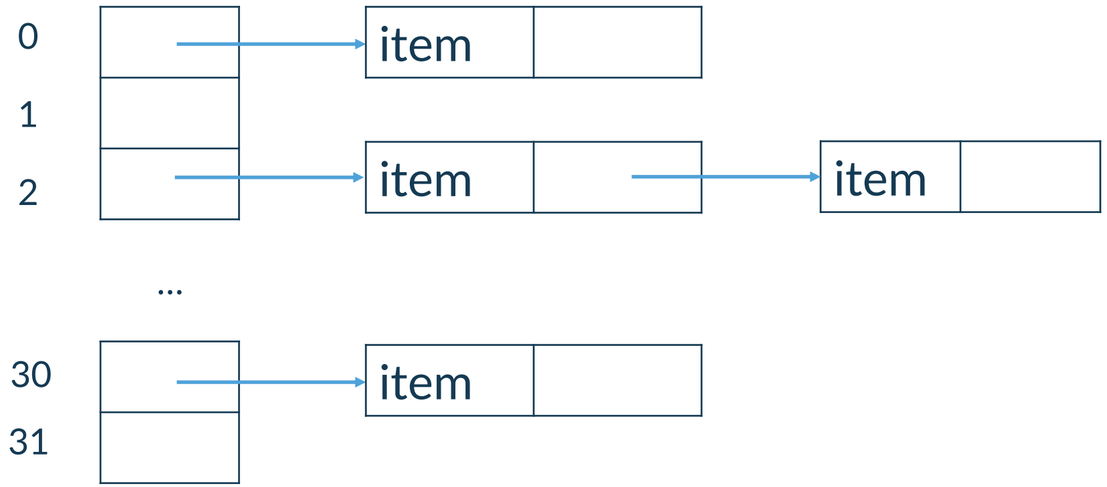
Creating unmodifiable sets

```
static <E> Set<E> of()
static <E> Set<E> of(E e1)
static <E> Set<E> of(E e1, E e2)
…
static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
static <E> Set<E> of(E... elements)
static <E> Set<E> copyOf(Collection<? extends E> coll)
```

The HashSet class

Set interface, backed by a hash table, it does not define any additional methods beyond those provided by its super classes and interfaces.

```
HashSet()
HashSet(Collection<? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```



HashSet has no predictable iteration order. LinkedHashSet allows insertion-order iteration over the map

The EnumSet class

A specialized Set implementation for use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created

```
public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)
public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)
public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)
public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)
public static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)
public static <E extends Enum<E>> EnumSet<E> of(E e)
public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2)
public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
public static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)
public static <E extends Enum<E>> EnumSet<E> range(E from, E to)
```

The SortedSet interface

```
public interface SortedSet<E> extends Set<E> {
    E first()
    E last()
    SortedSet<E> headSet(E toElement)
    SortedSet<E> subSet(E fromElement, E toElement)
    SortedSet<E> tailSet(E fromElement)
    Comparator<? super E> comparator()
}
```

The iterator return the elements using the specified order.

The NavigableSet interface

The NavigableSet interface extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.

```
public interface NavigableSet<E> extends SortedSet<E> {
```

`E ceiling(E e)` - Returns the least element in this set greater than or equal to the given element.

`E higher(E e)` - Returns the least element in this set strictly greater than the given element.

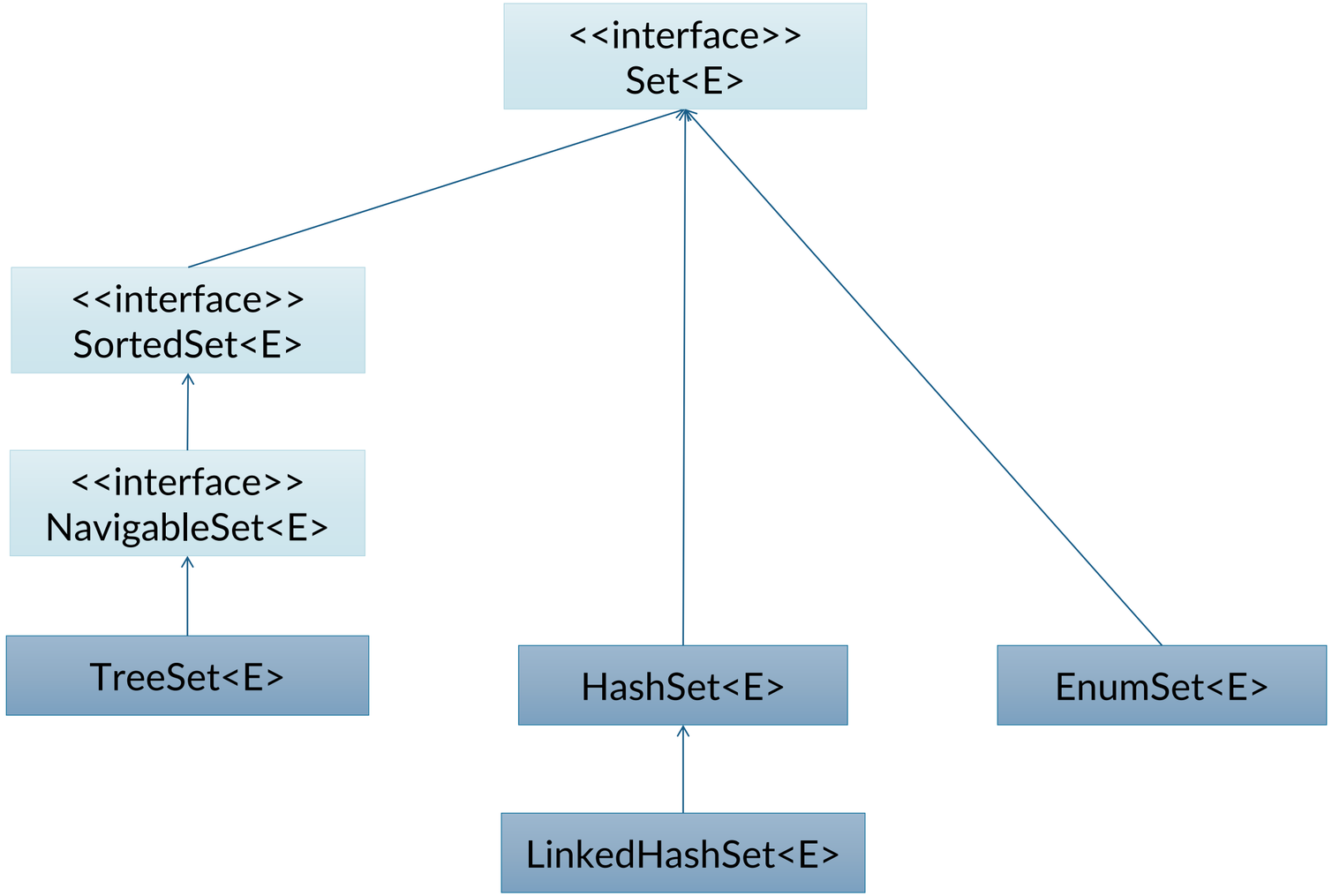
```
    E floor(E e)

    E lower(E e)

    E pollFirst()

    E pollLast()

    Iterator<E> descendingIterator()

    NavigableSet<E> descendingSet()

    …
}
```

The TreeSet class

The TreeSet class implements the NavigableSet and SortedSet interfaces, it based on a Red-Black tree implementation.

```
TreeSet( )

TreeSet(Collection<? extends E> c)

TreeSet(Comparator<? super E> comp)

TreeSet(SortedSet<E> ss)
```

Sets in summary

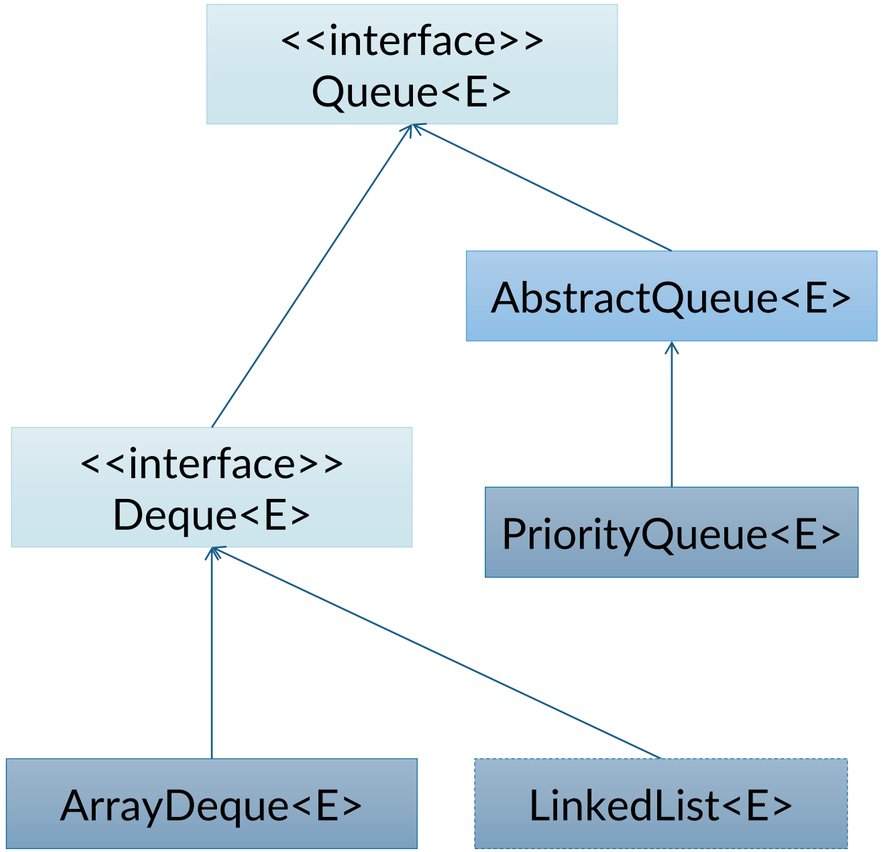


Queues

Queues

Queues are lists organized following a FIFO or LIFO criteria.

```
public interface Queue<E> extends Collection<E> {

    …

}
```



The Queue interface


```
public interface Queue<E> extends Collection<E> {
    boolean add(E e) throws an IllegalStateException
    boolean offer(E e)
    E element() throws NoSuchElementException
    E peek()
    E remove() throws NoSuchElementException
    E poll()
}
```

Limited capacity queue

The Deque interface

FIFO/LIFO

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	`addFirst(E e)`	`offerFirst(E e)`	`addLast(E e)`	`offerLast(E e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Methods from the Queue interface behave like a FIFO queue, we add elements from the tail and we remove them from the head

The ArrayDeque class

Resizable-array implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage.

```
public ArrayDeque()

public ArrayDeque(int numElements)

public ArrayDeque(Collection<? extends E> c)
```

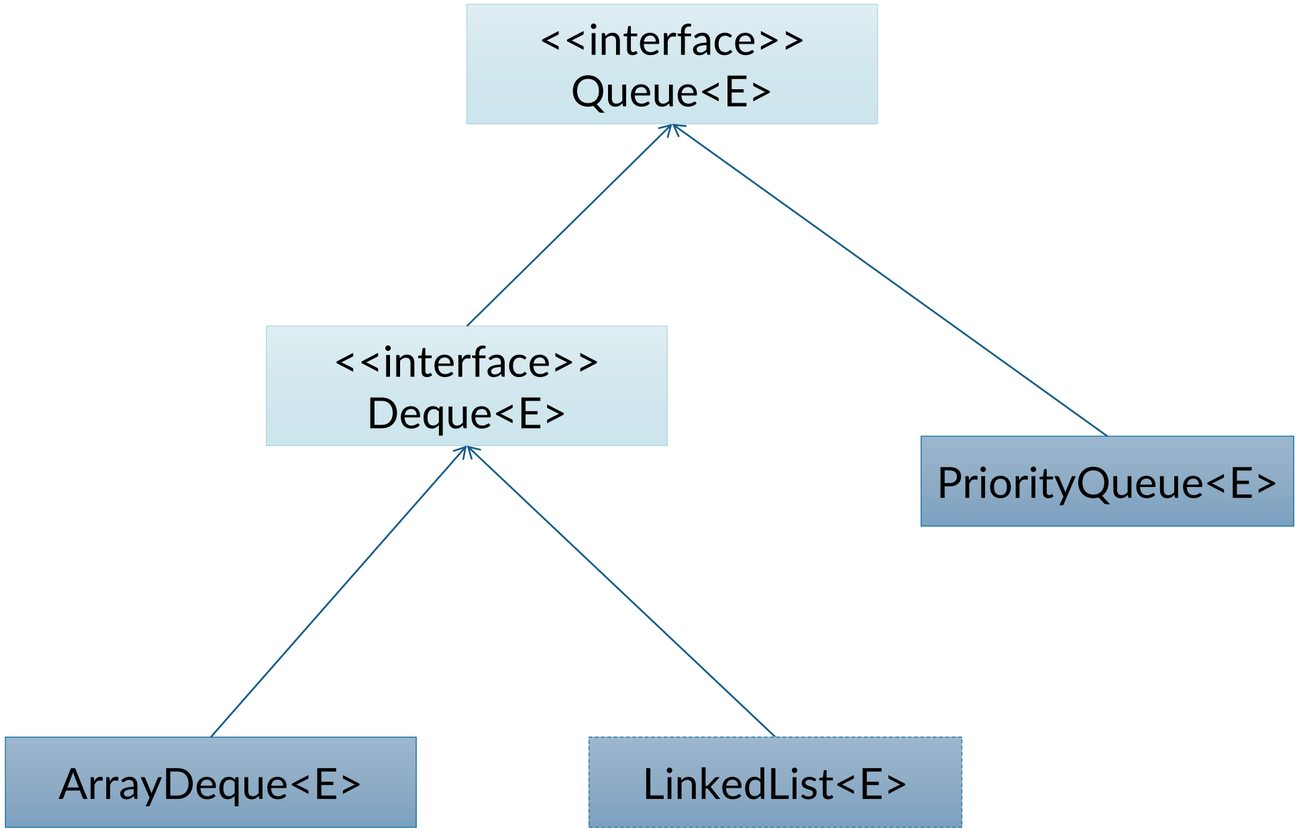
The PriorityQueue class

The PriorityQueue implements the Queue interface. It creates a queue that is prioritized based on the queue's comparator.

```
PriorityQueue( )

PriorityQueue(int capacity)

PriorityQueue(Comparator<? super E> comp)

PriorityQueue(int capacity, Comparator<? super E> comp)

PriorityQueue(Collection<? extends E> c)

PriorityQueue(PriorityQueue<? extends E> c)

PriorityQueue(SortedSet<? extends E> c)
```

Warning although you can iterate through a PriorityQueue using an iterator, the order of that iteration is undefined

Queues in summary



Examples with collections

Iterator

Every collection can return us an Iterator that enables us to cycle through the collection, obtaining or removing elements.

```
var set = Set.of(1, 2, 3, 4, 5);
for (Iterator<Integer> it = set.iterator(); it.hasNext(); ) {
    Integer item = it.next();
    if (Integer.valueOf(3).equals(item)) {
        it.remove();
    }
}
```

Lists can return a ListIterator that enables cycling through the collection in both directions

ConcurrentModificationException

```
var set = new HashSet<Integer>();
set.add(1);
set.add(2);
set.add(3);
set.add(4);
set.add(5);
for (Integer item : set) {
    if (Integer.valueOf(3).equals(item)) {
        set.remove(item);
    }
}
```

Altering a collection while cycling through the elements can cause a ConcurrentModificationException

Comparable

The Comparable interface is implemented by classes that want to define a “natural” order.

The String class implements Comparable, so String objects can be compared one with each other

```
public static void main(String[] args) {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    TreeSet<String> naturalOrder = new TreeSet<>(citiesOfFvg);

    System.out.println(naturalOrder);
}
```

```
[Gorizia, Pordenone, Trieste, Udine]
```

The result is the list of the cities
in the String lexicographical order

Comparator 1/3

Implementing the Comparator interface we can define an order for classes that doesn't implement Comparable or we can override the "natural" order of classes that already implements Comparable

```
public static void main(String[] args) {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    TreeSet<String> lengthOrder = new TreeSet<>(new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() - o2.length();
        }
    });
    lengthOrder.addAll(citiesOfFvg);

    System.out.println(lengthOrder);
}
```

```
[Udine, Trieste, Pordenone]
```

The result is the list of the cities in order of name length

What about Gorizia?

Comparator 2/3

```
Comparator<String> comparator = new Comparator<>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
};
TreeSet<String> triesteOnly = new TreeSet<>(comparator);
triesteOnly.add("Trieste");

System.out.println(triesteOnly.contains("Gorizia"));
```

The answer is true because the order relation defined by the comparator is used as equivalence relation.

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface.
(See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

Comparator 3/3

```
public static void main(String[] args) {
    Set<String> citiesOfFvg = Set.of("Trieste", "Udine", "Gorizia", "Pordenone");

    Comparator<String> comparator = new Comparator<>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.length() == o2.length() ? o1.compareTo(o2) : o1.length() - o2.length();
        }
    };

    TreeSet<String> lengthOrder = new TreeSet<>(comparator);
    lengthOrder.addAll(citiesOfFvg);

    System.out.println(lengthOrder);
}
```

```
[Udine, Gorizia, Trieste, Pordenone]
```

Gorizia is back!

Object.equals()

```
public boolean equals(Object obj)
```

Indicates whether some other object is “equal to” this one

The equals method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
- It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified
- For any non-null reference value x, x.equals(null) should return false

An equivalence relation partitions the elements it operates on into *equivalence classes*; all the members of an equivalence class are equal to each other. Members of an equivalence class are substitutable for each other, at least for some purposes

The default implementation is the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true). In other words, under the reference equality equivalence relation, each equivalence class only has a single element

Equals and ordering

The implementations of the methods of the Comparable and Comparator interfaces must be consistent with equals

The “natural” ordering for a class C (implementing Comparable) is said to be *consistent with equals* if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C.

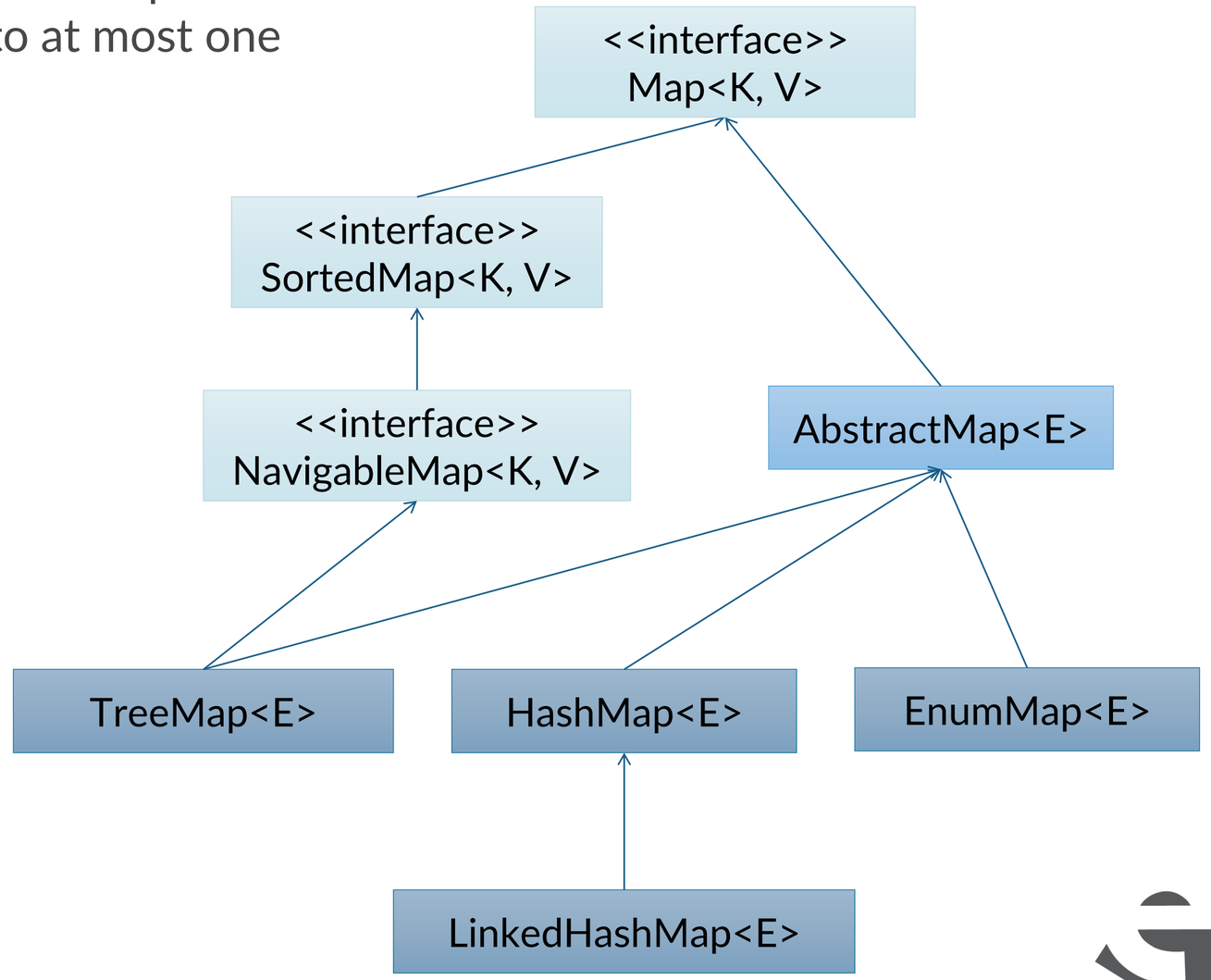
Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

Maps

Maps

A map is an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

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



Querying a map

```
V get(Object key)

boolean containsKey(Object key)

boolean containsValue(Object value)

default V getOrDefault(Object key, V defaultValue)

boolean isEmpty()

int size()

Set<K> keySet()

Collection<V> values()

Set<Map.Entry<K,V>> entrySet()
```

Modifying a map

```
V put(K key, V value)
V remove(Object key)
default V replace(K key, V value)
void putAll(Map<? extends K,? extends V> m)
default V putIfAbsent(K key, V value)
default boolean replace(K key, V oldValue, V newValue)
default boolean remove(Object key, Object value)
boolean isEmpty()
void clear()
```

All these methods are optional, and they throw UnsupportedOperationException if the collection is unmodifiable.

Creating unmodifiable maps

```
static <K,V> Map<K,V> copyOf(Map<? extends K,? extends V> map)
static <K,V> Map.Entry<K,V> entry(K k, V v)
static <K,V> Map<K,V> of()
static <K,V> Map<K,V> of(K k1, V v1)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)
…
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, … K k10, V v10)
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)
```

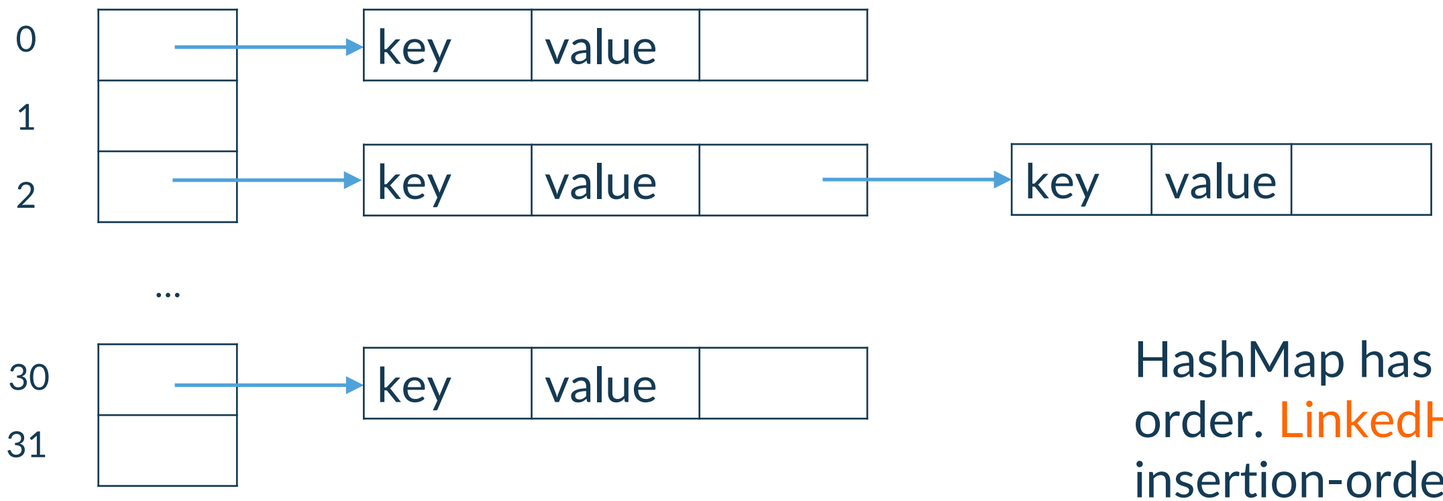
The Map interface

```
default V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
default V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)
default V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
default void forEach(BiConsumer<? super K,? super V> action)
default V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)
default void replaceAll(BiFunction<? super K,? super V,? extends V> function)
```

HashMap

Implements the Map interface, backed by a hash table, it does not define any additional methods beyond those provided by its super classes and interfaces.

```
HashMap()

HashMap(Collection<? extends E> c)

HashMap(int capacity)

HashMap(int capacity, float fillRatio)
```



HashMap has no predictable iteration order. LinkedHashMap allows insertion-order iteration over the map

Object.hashCode()

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application
- If two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result
- It is *not* required that if two objects are unequal according to the equals method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables

HashCode and equals 1/4

Suppose we want to define a File class to represent file systems objects

And we want to store the file sizes into a Map, so that we can retrieve that information from the map

```
public class File {

    private String name;

    public File(String name) {
        this.name = name;
    }
}
```

```
public static void main(String[] args) {
    var fileSizes = new HashMap<File, Long>();
    fileSizes.put(new File("C:\\Users\\pvercesi\\lesson8.pdf"), 342340L);
    fileSizes.put(new File("C:\\Users\\pvercesi\\lesson9.pdf"), 512956L);

    File file = new File("C:\\Users\\pvercesi\\lesson8.pdf");
    System.out.println(fileSizes.get(file));
}
```

This example doesn't work as expected

Why?

HashCode and equals 2/4

```
public class File {
    private String name;

    public File(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        File file = (File) o;
        return Objects.equals(name, file.name);
    }

    @Override
    public int hashCode() {
        return name.hashCode();
    }
}
```

HashCode and equals 3/4

Now we want
to rename a File

```
public class File {

    private String name;

    public File(String name) {
        this.name = name;
    }

    public void rename(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    …
}
```

HashCode and equals 4/4

```
public static void main(String[] args) {
    var fileSizes = new HashMap<File, Long>();
    fileSizes.put(new File("C:\\Users\\pvercesi\\Desktop\\lesson8.pdf"), 342340L);
    fileSizes.put(new File("C:\\Users\\pvercesi\\Desktop\\lesson9.pdf"), 512956L);

    String fromName = "C:\\Users\\pvercesi\\Desktop\\lesson8.pdf";
    String toName = "C:\\Users\\pvercesi\\Desktop\\lesson8a.pdf";

    for (File file : fileSizes.keySet()) {
        if (file.getName().equals(fromName)) {
            file.rename(toName);
        }
        break;
    }

    File file = new File("C:\\Users\\pvercesi\\Desktop\\lesson8a.pdf");
    System.out.println(fileSizes.get(file));
}
```

After renaming, the hash code is changed but the object position in the hash table is not updated!
The API of the file object is flawed, we cannot use this class as key in hash based collections

Fortunately, the java.io.File class of the Java API, doesn't work in this way ☺

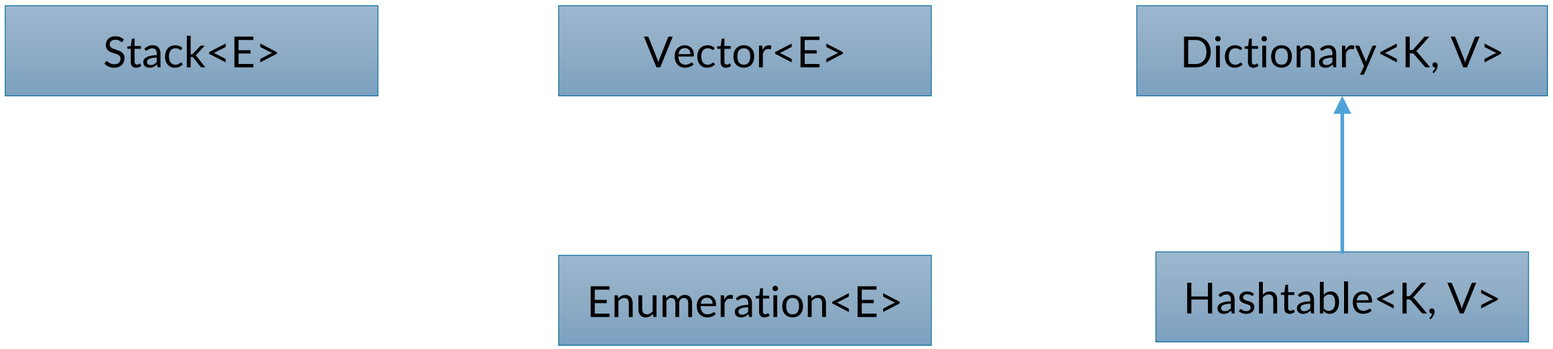
Summary

The Collections class

The java.util.Collections class provide several methods to work with collections

- factory methods to get empty collections
- factory methods to build unmodifiable collections
- factory methods for synchronized collections
- search in ordered lists
- copy, sort, rotate, shuffle, and fill lists
- count and replace occurrences
- find min and max
- ..

Legacy classes



They are part of the old Java API, you should not use them. If you are using an API that uses these classes, maybe there is more a modern implementation.

Summary of implementations

Data structure	List	Queue	Set	Map
Array based	ArrayList	ArrayDeque		
Linked	LinkedList		LinkedHashSet	LinkedHashMap
Hash based			HashSet	HashMap
Tree based			TreeSet	TreeMap
Bit-set based			EnumSet	EnumMap
Priority queue		PriorityQueue		

Assignment

Assignment

Write a class (or a set of classes) that given a string it produces a Term Frequency table.
Consider the option to provide a list of stop words, normalization, etc.
Provide an option to print the table in alphabetical order and by frequency.

"Term frequency (TF) means how often a term occurs in a document. In the context of natural language, terms correspond to words or phrases …"

Term	Frequency
english	8
language	7
words	12

…

input	7
cactus	1
fireworks	3

Thank you!

esteco.com