Java Collection Framework (JCF)

Problem

Problem

- Конечный размер массива
- Удаление и добавление элементов массива
- Реализаций динамических структур данных таких как:
  - связные списки
  - деревья
  - т.д.

Solution

Collections

Collections

- A data structure that allows you to store objects in a convenient way
- Contains a set of methods for manipulating data
- All collections are objects
- Cannot store primitive data types

Collection Framework

- **Collection framework** (**каркас коллекций**) — это унифицированная архитектура для представления и манипулирования коллекциями.
- Collection framework содержит:
  - **Interfaces** (**Интерфейсы**)
  - **Implementations** (**Реализации**)
  - **Algorithms** (**Алгоритмы**)

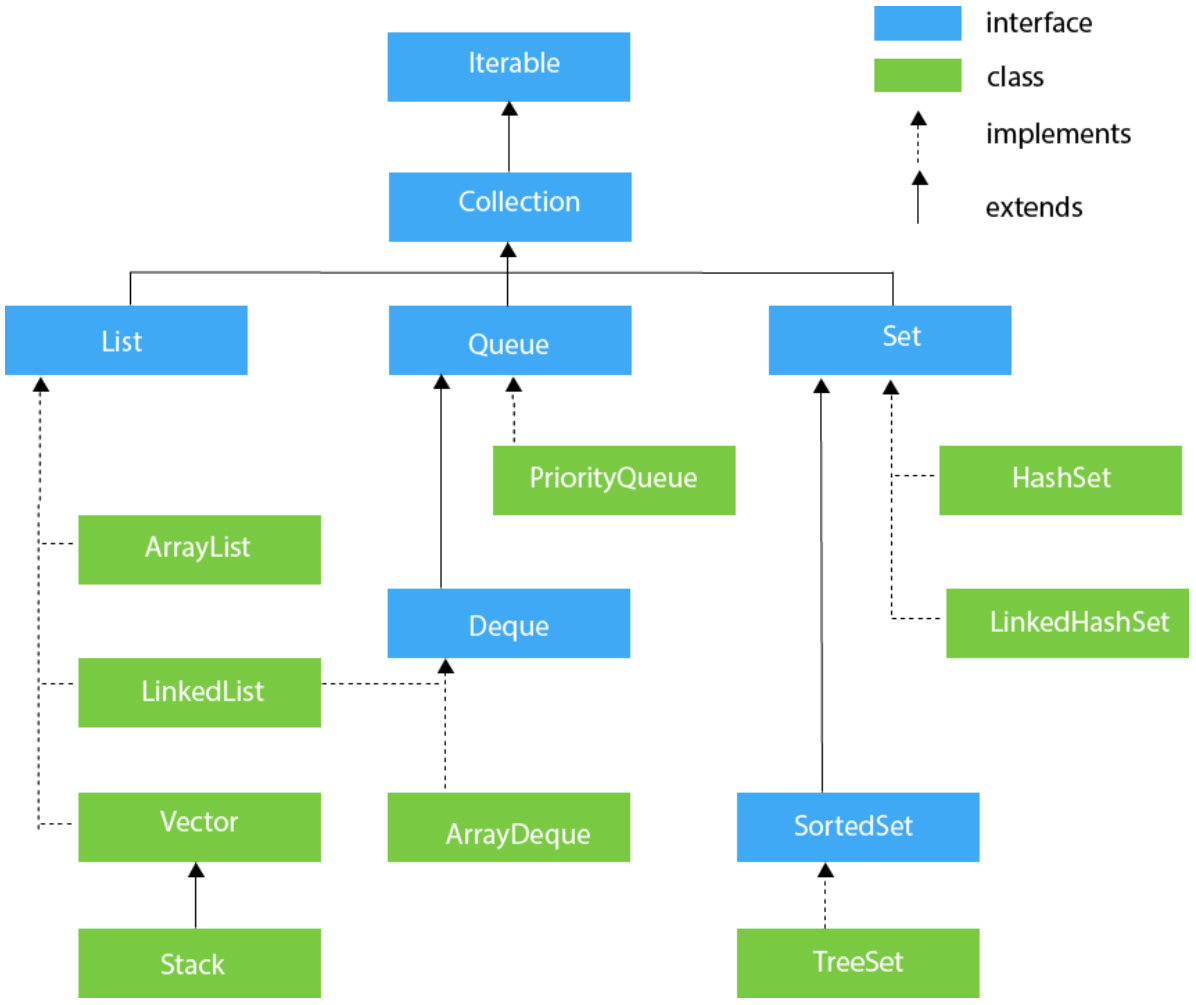
Implements Collections

- **Java Collection Framework**
- **Guava** (Google Collection Library)
- **Trove library** (primitive JCF)
- **PCJ** (Primitive Collections for Java)
- **Custom collection** (not recommend)

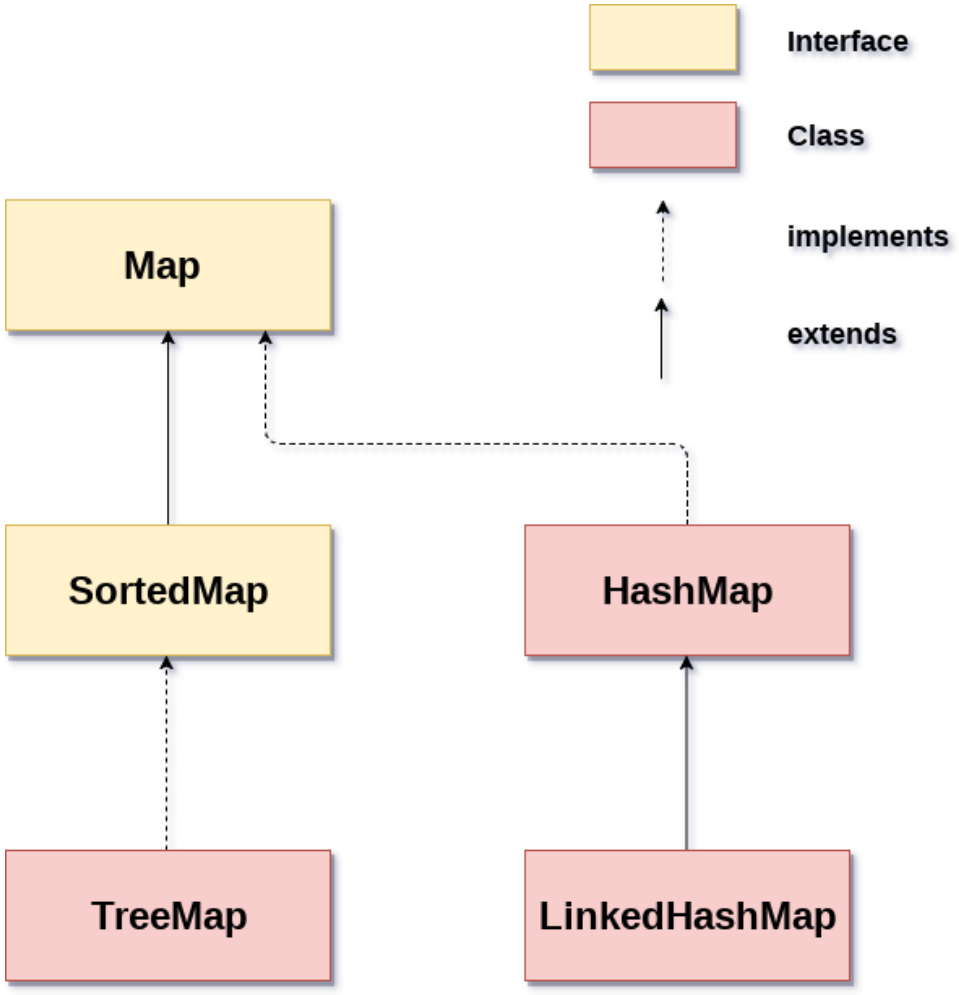
Deprecated Classes

- `Enumeration`
- `Vector`
- `Stack`
- `Dictionary`
- `Hashtable`

Collection Hierarchy

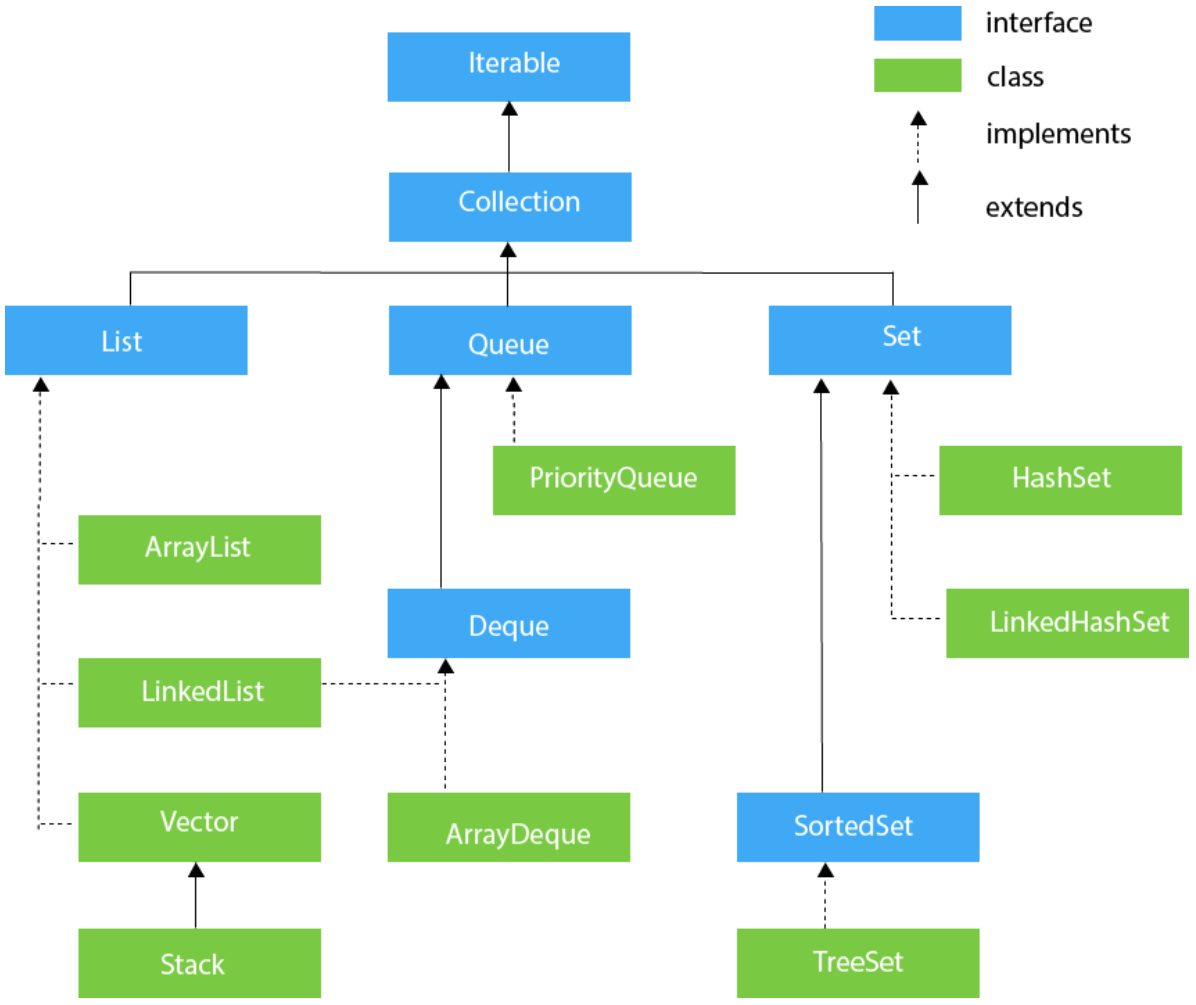


Map Hierarchy



Interface `Collection<E>`

Collection Hierarchy



Interface `Collection<E>`

- `Collection<E>` - вершина иерархии остальных коллекций
- `List<E>` - специализирует коллекции для обработки списков
- `Set<E>` - специализирует коллекции для обработки множеств, содержащих уникальные элементы
- `Queue<E>` - коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки
- `Map<K, V>` - карта отображения вида *ключ-значение*;

Интерфейсы позволяют манипулировать коллекциями независимо от деталей конкретной реализации, реализуя тем самым принцип полиморфизма.

Methods

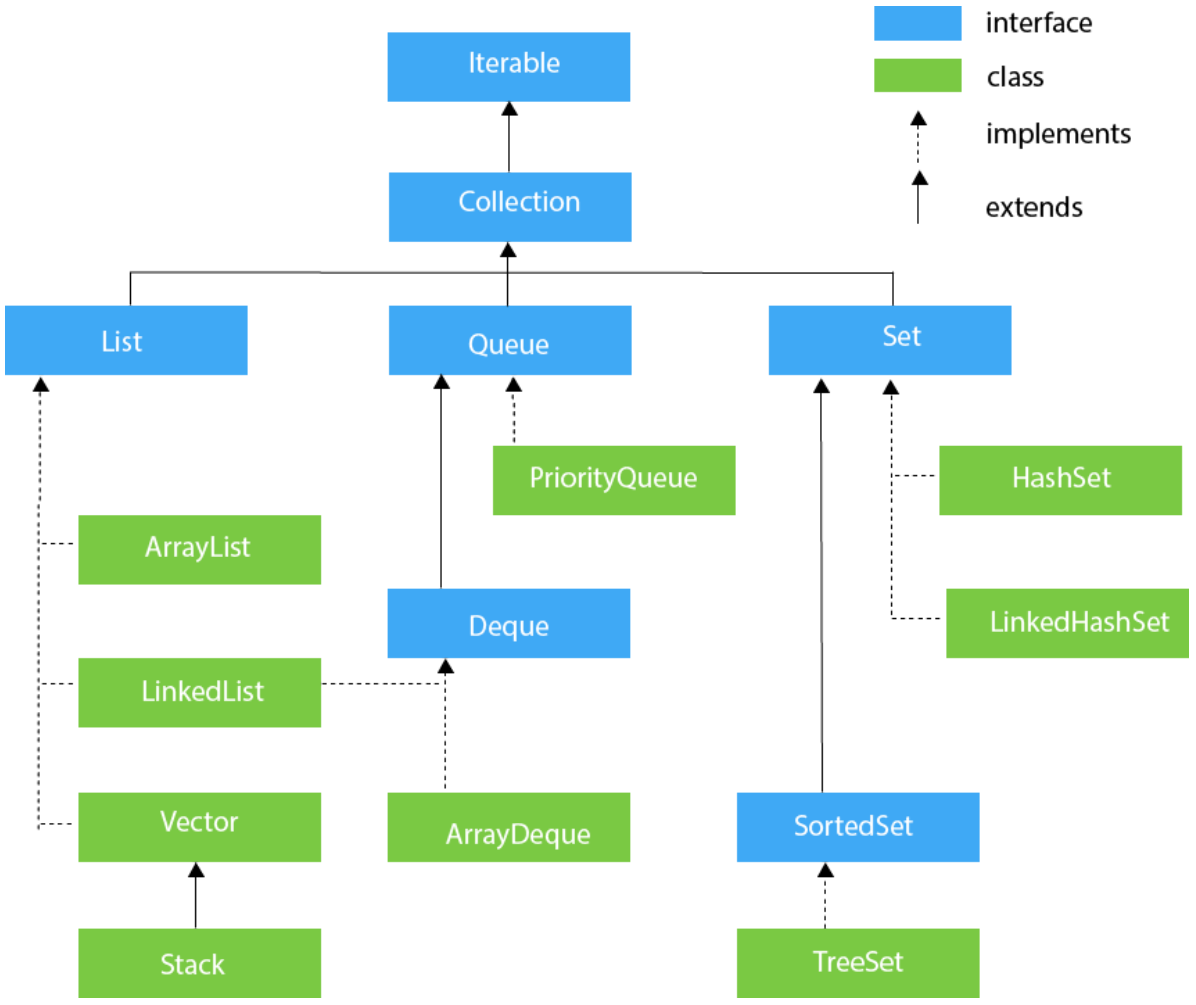
- `add(E item): boolean`
- `addAll(Collection<? extends E> col): boolean`
- `clear(): void`
- `contains(Object item): boolean`
- `isEmpty(): boolean`
- `iterator(): Iterator<E>`
- `remove(Object item): boolean`
- `removeAll(Collection<?> col): boolean`

Methods

- `retainAll(Collection<?> col): boolean`
- `size(): int`
- `toArray(): Object[]`

Interface `Iterator<E>`

Collection Hierarchy



Interface `Iterator<E>`

Iterator (**Итератор**) — это объект, предназначенный для обхода коллекции.

Methods

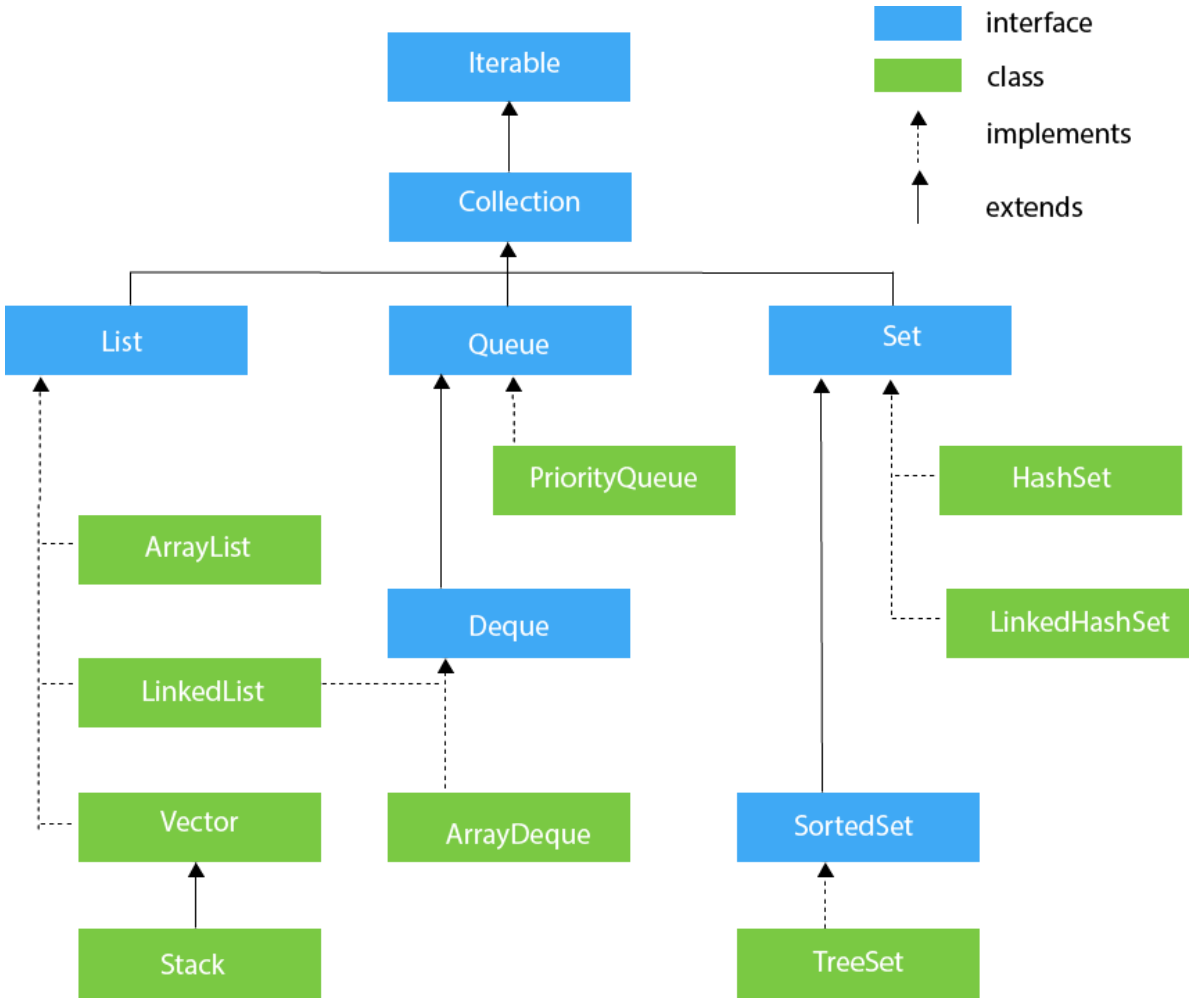
- `next(): E`
- `hasNext(): boolean`
- `remove(): void`

Example

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> states = new ArrayList<String>();
        states.add("Germany");
        states.add("France");
        states.add("Italy");
        states.add("Spain");

        Iterator<String> iter = states.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

Interface `List<E>`

Collection Hierarchy



Interface `List<E>`

- Интерфейс `List<E>` служит для описания списков.
- Список может содержать повторяющиеся элементы.
- `List<E>` сохраняет последовательность добавления элементов и позволяет осуществлять доступ к элементу по индексу.

Methods

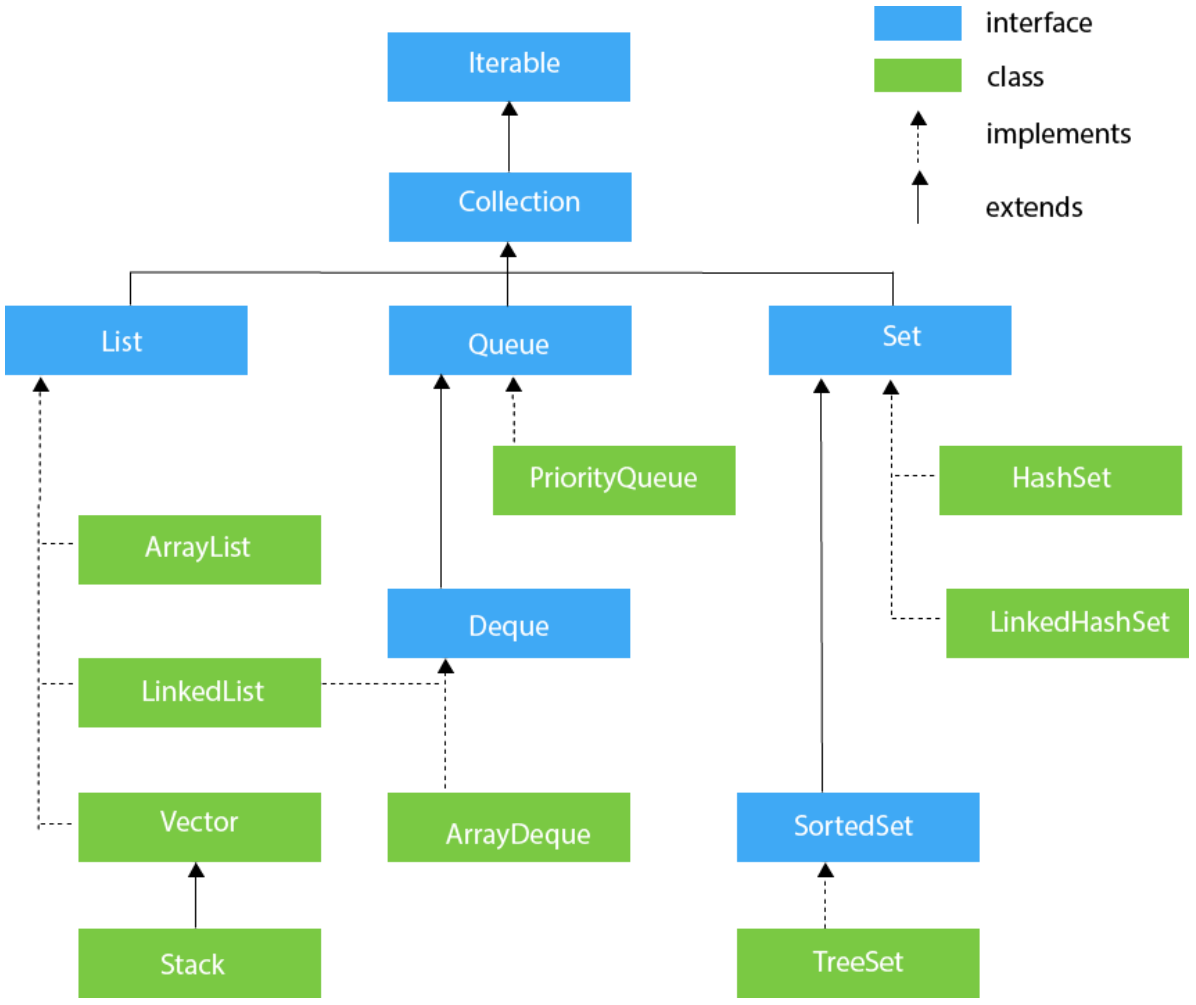
- `add(int index, E obj): void`
- `addAll(int index, Collection<? extends E> col): boolean`
- `get(int index): E`
- `indexOf(Object obj): int`
- `lastIndexOf(Object obj): int`
- `listIterator(): ListIterator<E>`
- `static <E> of(E …): List<E>`
- `remove(int index): E`

Methods

- `set(int index, E obj): E`
- `sort(Comparator<? super E> comp): void`
- `subList(int start, int end): List<E>`

Interface `ListIterator<E>`

Collection Hierarchy



Methods

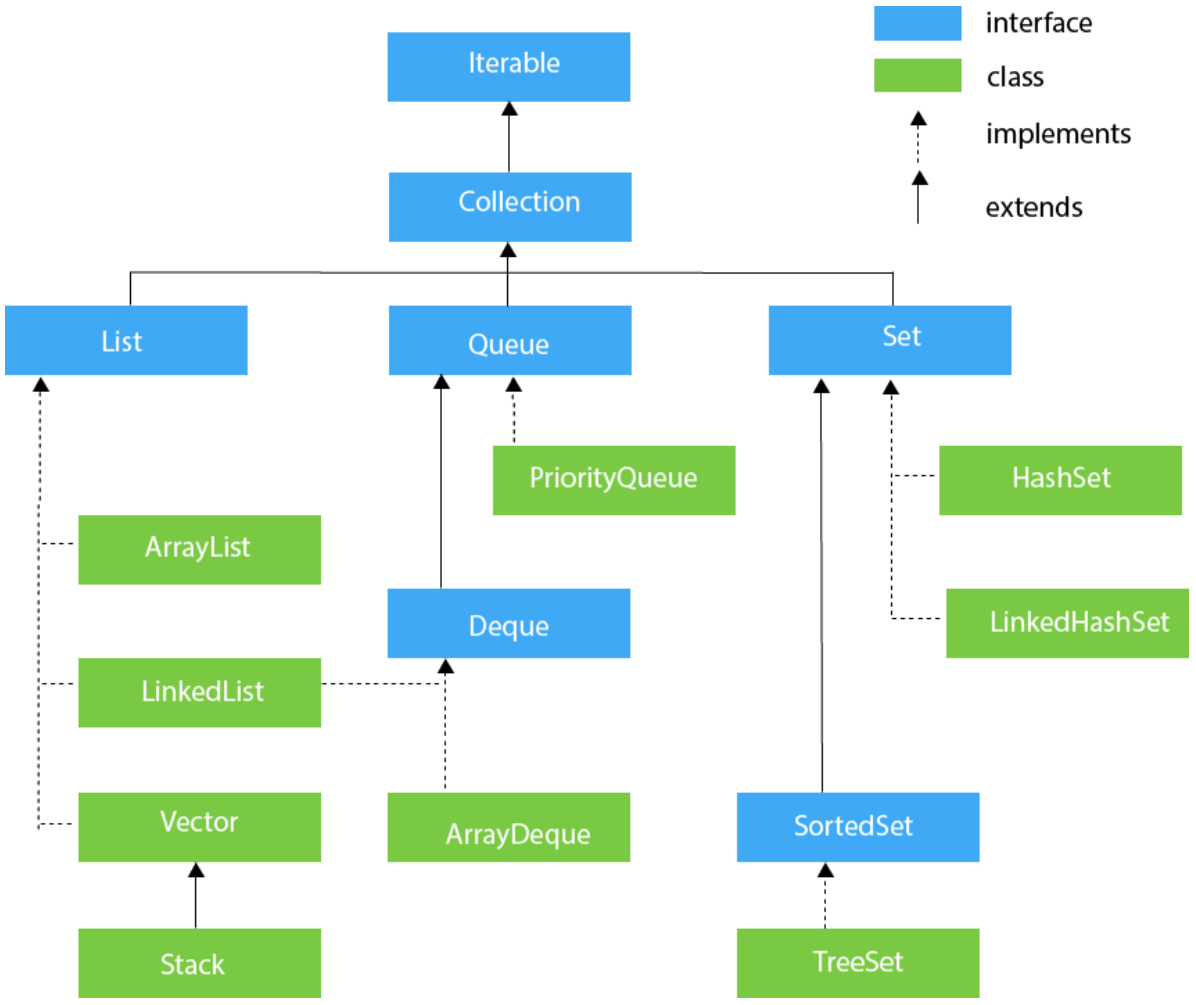
- `add(E obj): void`
- `hasNext(): boolean`
- `hasPrevious(): boolean`
- `next(): E`
- `previous(): E`
- `nextIndex(): int`
- `previousIndex(): int`
- `remove(): void`
- `set(E obj): void`

Example

```
import java.util.ArrayList;
import java.util.ListIterator;

public class ListIteratorExample {
    public static void main(String[] args) {
        ArrayList<String> states = new ArrayList<String>();
        states.add("Germany");
        states.add("France");
        states.add("Italy");
        states.add("Spain");

        ListIterator<String> listIter = states.listIterator()

        while (listIter.hasNext()) {
            System.out.println(listIter.next());
        }

        listIter.set("Португалия");
```

Class `ArrayList<E>`

Collection Hierarchy



Class `ArrayList<E>`

- `ArrayList<E>` - список на базе массива.
- Аналогичен `Vector<E>` за исключением потокобезопасности.
- Можно использовать для:
  - “Бесконечный” массив
  - Стек

Constructors

- `ArrayList()`
- `ArrayList(Collection <? extends E> col)`
- `ArrayList(int capacity)`

Example

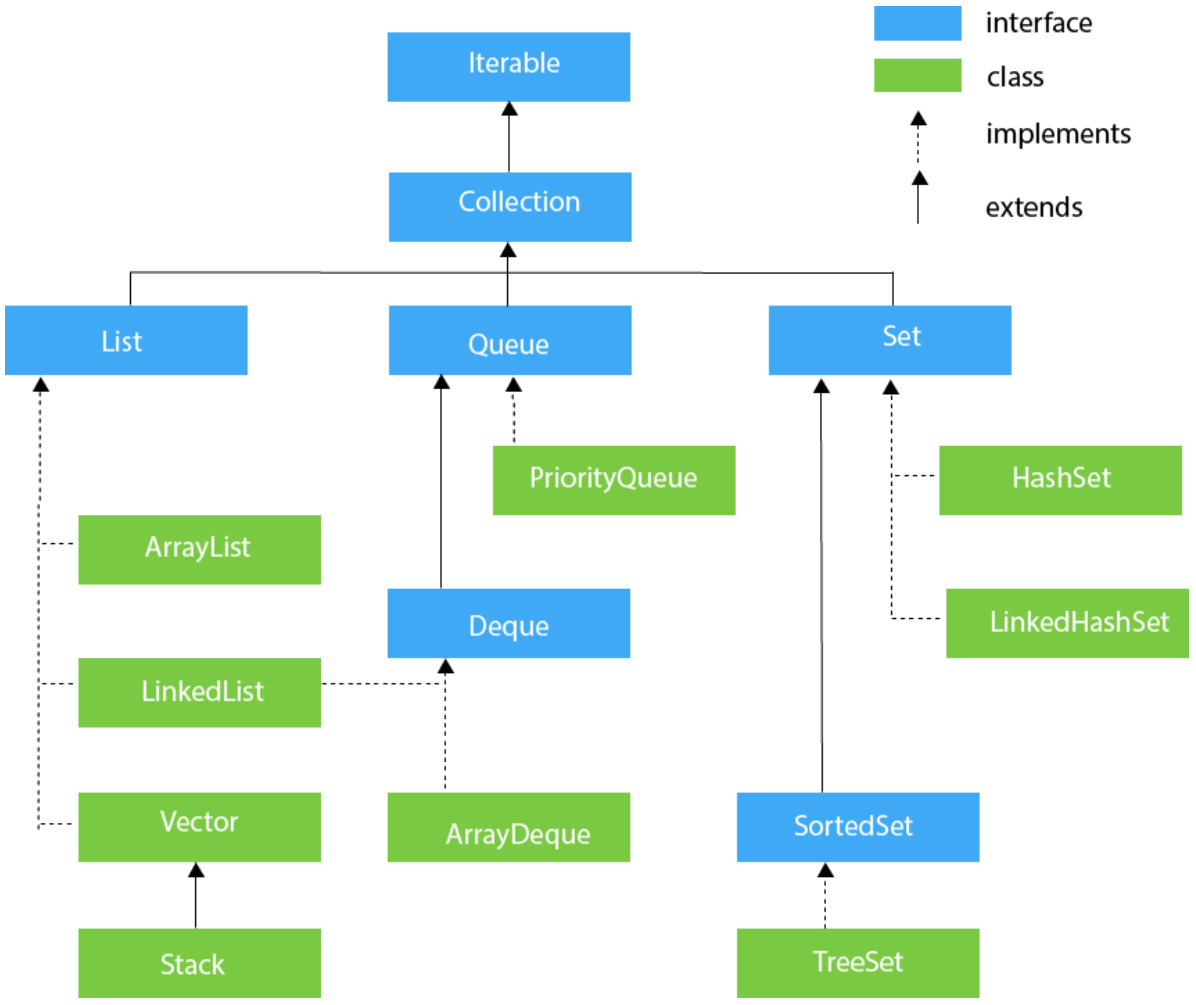
```
import java.util.ArrayList;
import java.util.Iterator;

class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Ravi");
        list.add("Vijay");
        list.add("Ravi");
        list.add("Ajay");

        Iterator itr = list.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Profit

- Плюсы
  - Быстрый доступ по индексу
  - Быстрая вставка и удаление элементов с конца
- Минусы
  - Медленная вставка и удаление элементов

Interface Queue<E>

Collection Hierarchy



Interface `Queue<E>`

- **Queue<E>** (**Очередь**) — хранилище элементов для обработки.
- Свойства очередей:
  - Порядок выдачи элементов определяется конкретной реализацией
  - Очереди не могут хранить `null`
  - У очереди может быть ограничен размер

Methods

- `element(): E`
- `offer(E obj): boolean`
- `peek(): E`
- `poll(): E`
- `remove(): E`

Class `PriorityQueue<E>`

Class `PriorityQueue<E>`

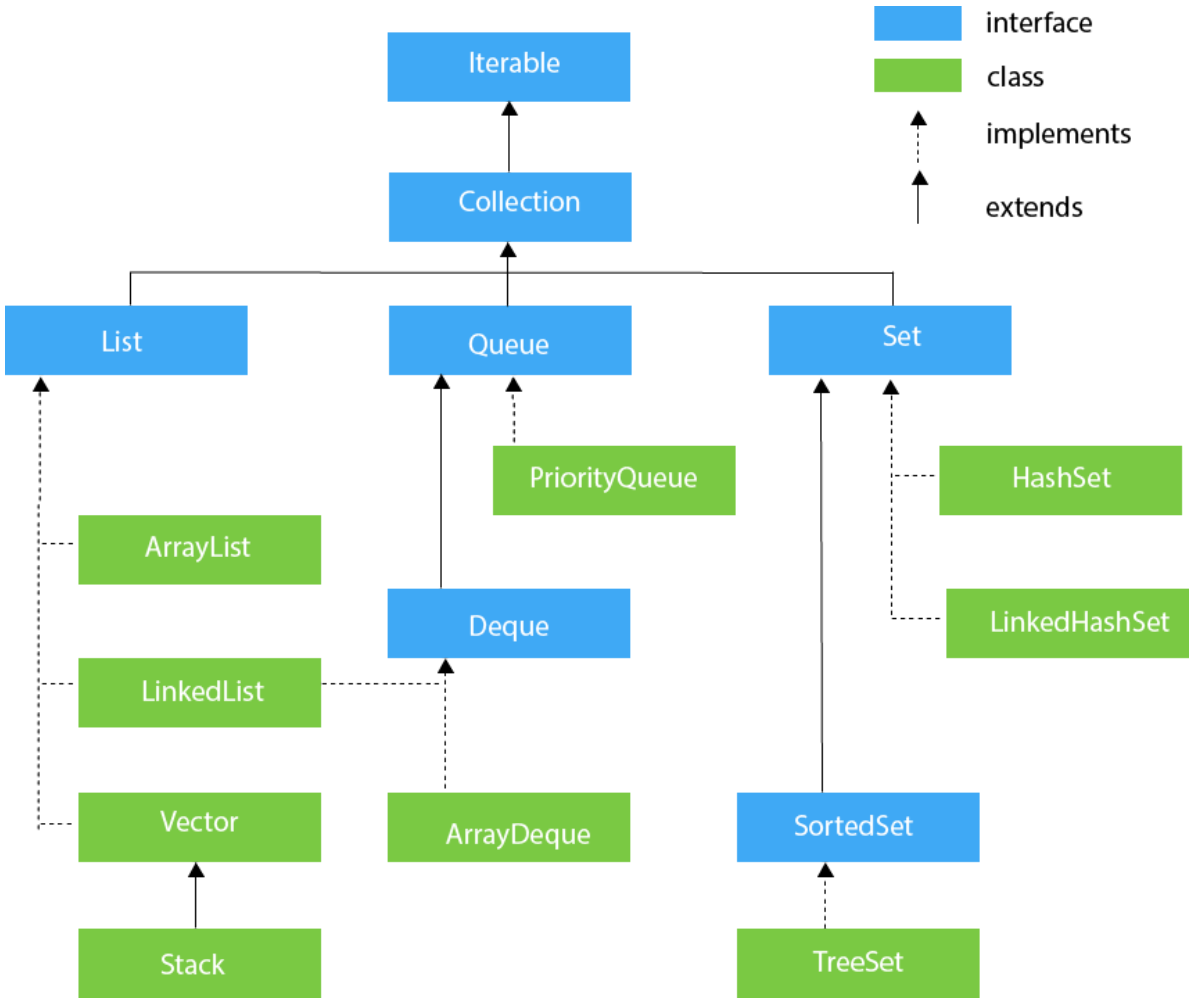
- `PriorityQueue<E>` - это класс очереди с приоритетами.
- По умолчанию очередь с приоритетами размещает элементы согласно естественному порядку сортировки используя `Comparable`.
- Элементу с наименьшим значением присваивается наибольший приоритет.
- Если несколько элементов имеют одинаковый наивысший элемент – связь определяется произвольно.
- Также можно указать специальный порядок размещения, используя `Comparator`.

Constructors

- `PriorityQueue()` *default* `capacity 11`
- `PriorityQueue(Collection<? extends E> c)`
- `PriorityQueue(int initialCapacity)`
- `PriorityQueue(int initialCapacity,`
  `Comparator<? super E> comparator)`
- `PriorityQueue(PriorityQueue<? extends E> c)`
- `PriorityQueue(SortedSet<? extends E> c)`

Interface Deque<E>

Collection Hierarchy



Methods

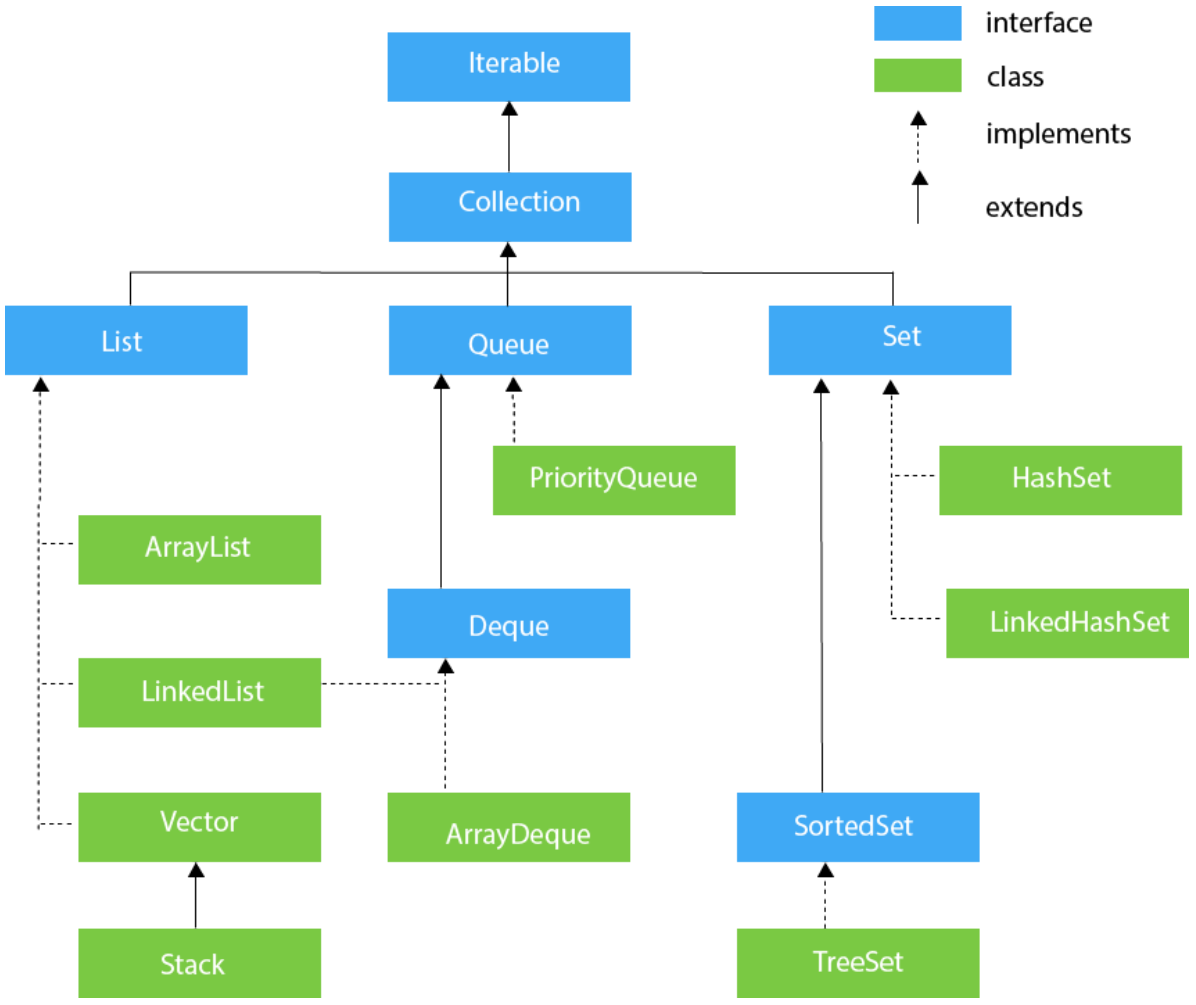
- `addFirst(E obj): void`
- `addLast(E obj): void`
- `getFirst(): E`
- `getLast(): E`
- `offerFirst(E obj): boolean`
- `offerLast(E obj): boolean`
- `peekFirst(): E`
- `peekLast(): E`

Methods

- `pollFirst(): E`
- `pollLast(): E`
- `pop(): E`
- `push(E element): void`
- `removeFirst(): E`
- `removeLast(): E`
- `removeFirstOccurrence(Object obj): boolean`
- `removeLastOccurrence(Object obj): boolean`

Class `ArrayDeque<E>`

Collection Hierarchy



Constructors

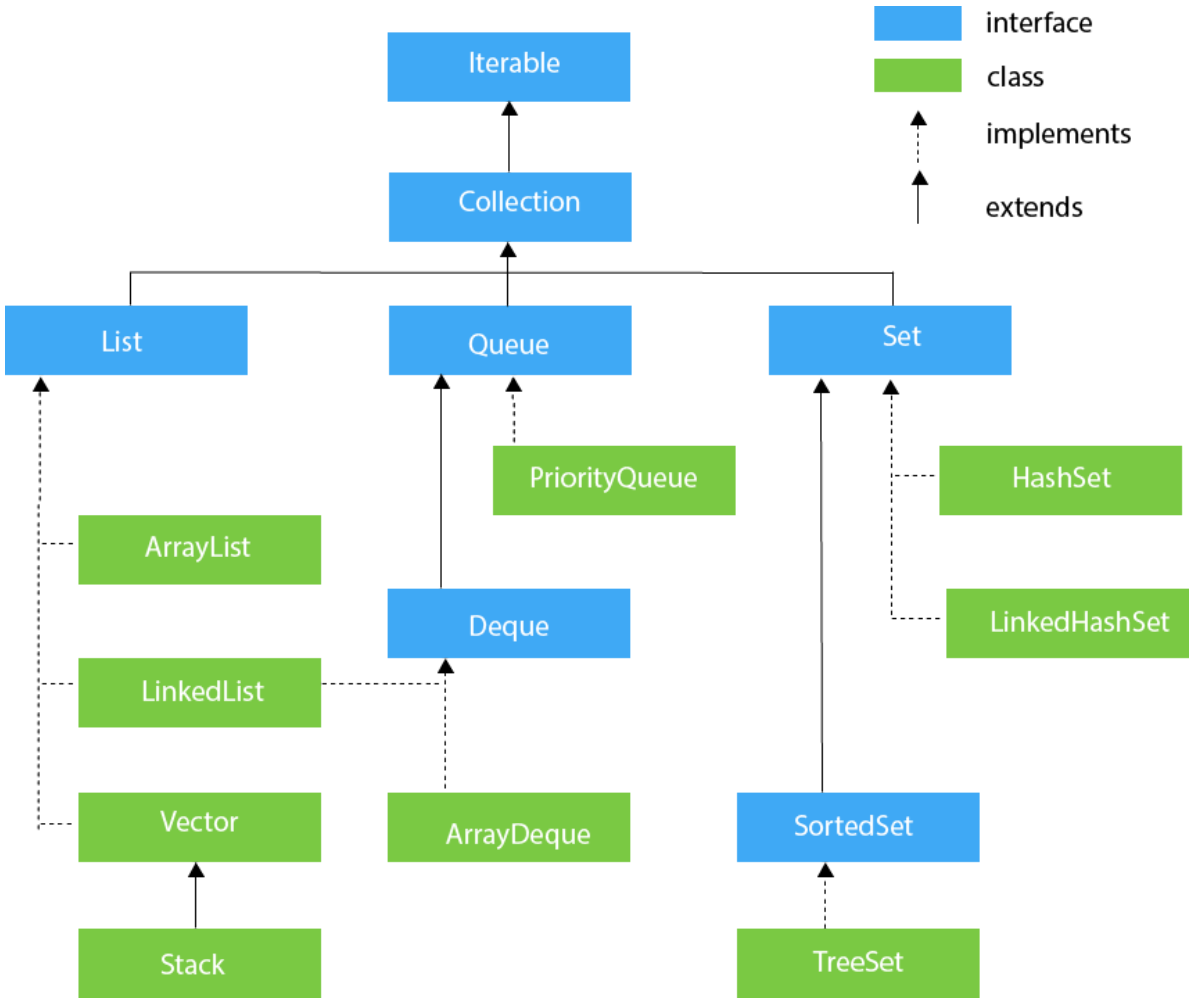
- `ArrayDeque()`
- `ArrayDeque(Collection<? extends E> col)`
- `ArrayDeque(int capacity)` *default* `16`

Example

```
import java.util.ArrayDeque;
import java.util.Deque;

public class ArrayDequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<String>();
        deque.add("Ravi");
        deque.add("Vijay");
        deque.add("Ajay");
        for (String str : deque) {
            System.out.println(str);
        }
    }
}
```

Class `LinkedList<E>`

Collection Hierarchy



Class `LinkedList<E>`

- `LinkedList<E>` - связный список.
- Рекомендуется использовать, если необходимо часто добавлять элементы в начало списка или удалять внутренний элемент списка.
- Можно использовать для:
  - Стек
  - Очередь
  - Двухсторонняя очередь

Constructors

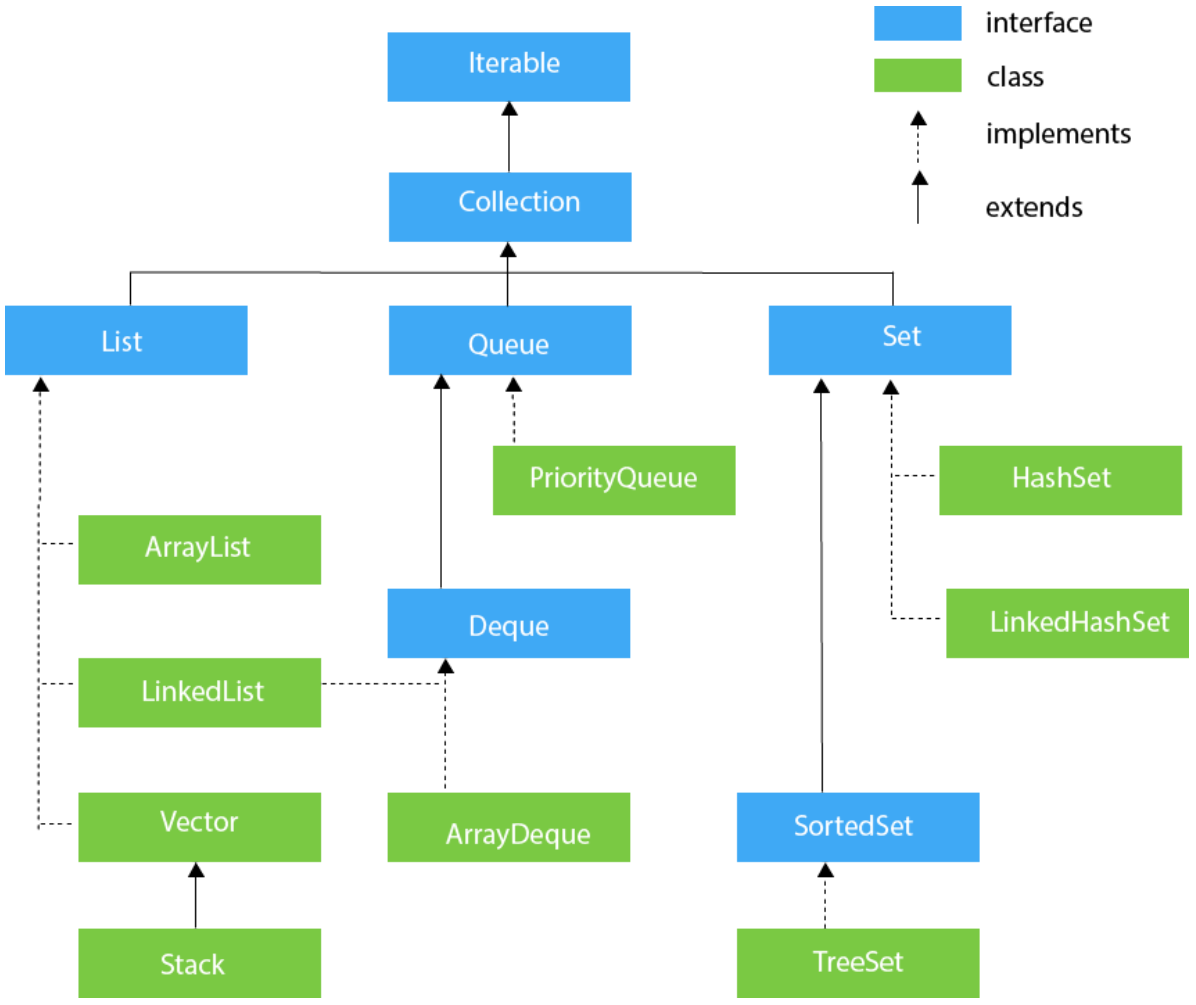
- `LinkedList()`
- `LinkedList(Collection<? extends E> col)`

Profit

- Плюсы:
  - Быстрое добавление и удаление элементов
- Минусы:
  - Медленный доступ по индексу

Interface `Set<E>`

Collection Hierarchy

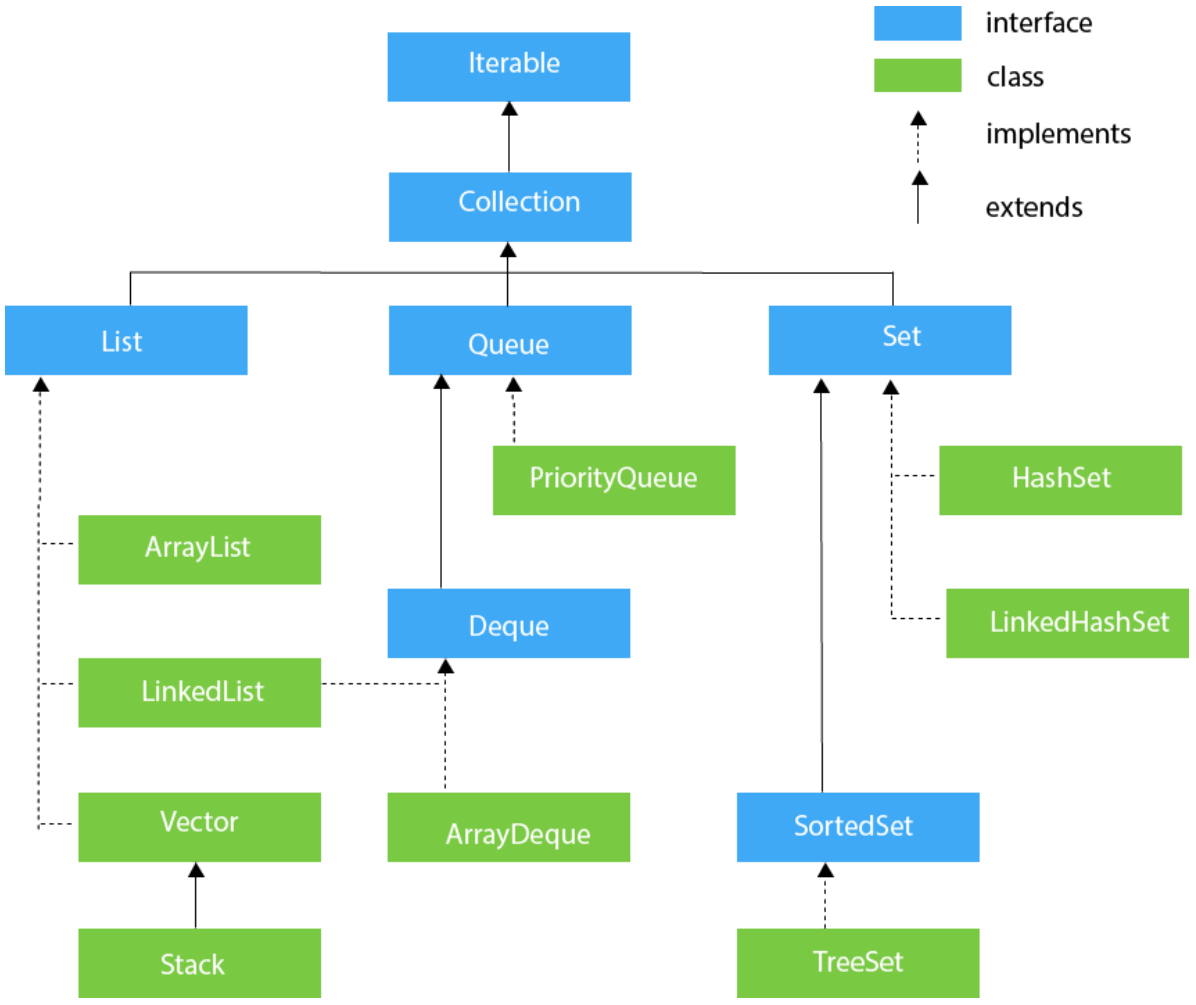


Interface `Set<E>`

- **Set<E>** (**Множество**) — коллекция без повторяющихся элементов.
- Интерфейс `Set<E>` содержит методы, унаследованные `Collection<E>` и добавляет запрет на дублирующийся элементы.

Class `HashSet<E>`

Collection Hierarchy



Class `HashSet<E>`

`HashSet<E>` - неупорядоченное множество на основе хэш кода.

Constructors

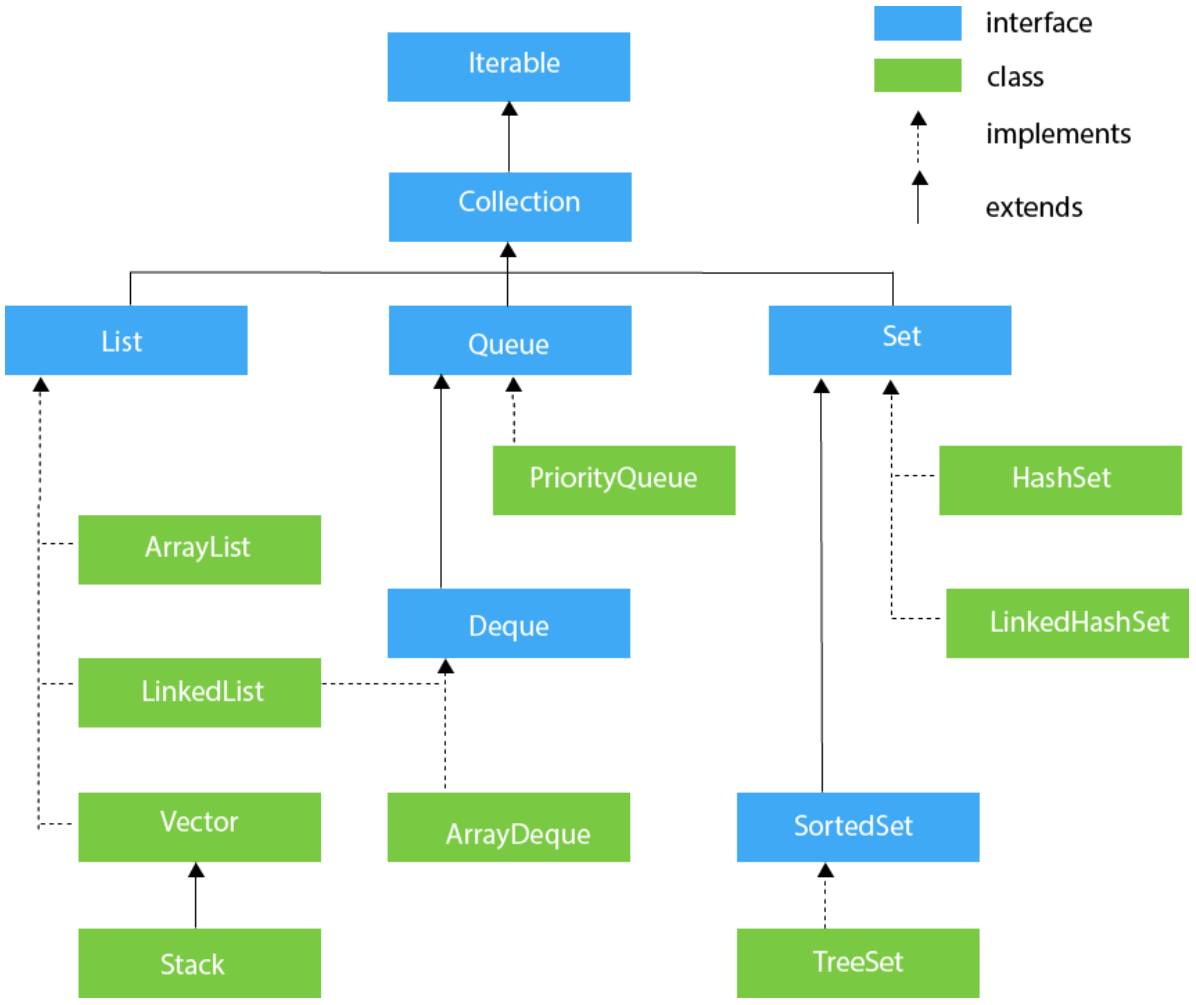
- `HashSet()`
- `HashSet(Collection<? extends E> col)`
- `HashSet(int capacity)`, где *default* `16`
- `HashSet(int capacity, float koef)`, где `koef` [`0.0`; `1.0`]

Example

```
import java.util.HashSet;
import java.util.Iterator;

class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<String>();
        set.add("Ravi");
        set.add("Vijay");
        set.add("Ravi");
        set.add("Ajay");
        Iterator<String> itr = set.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Interface `SortedSet<E>`

Collection Hierarchy

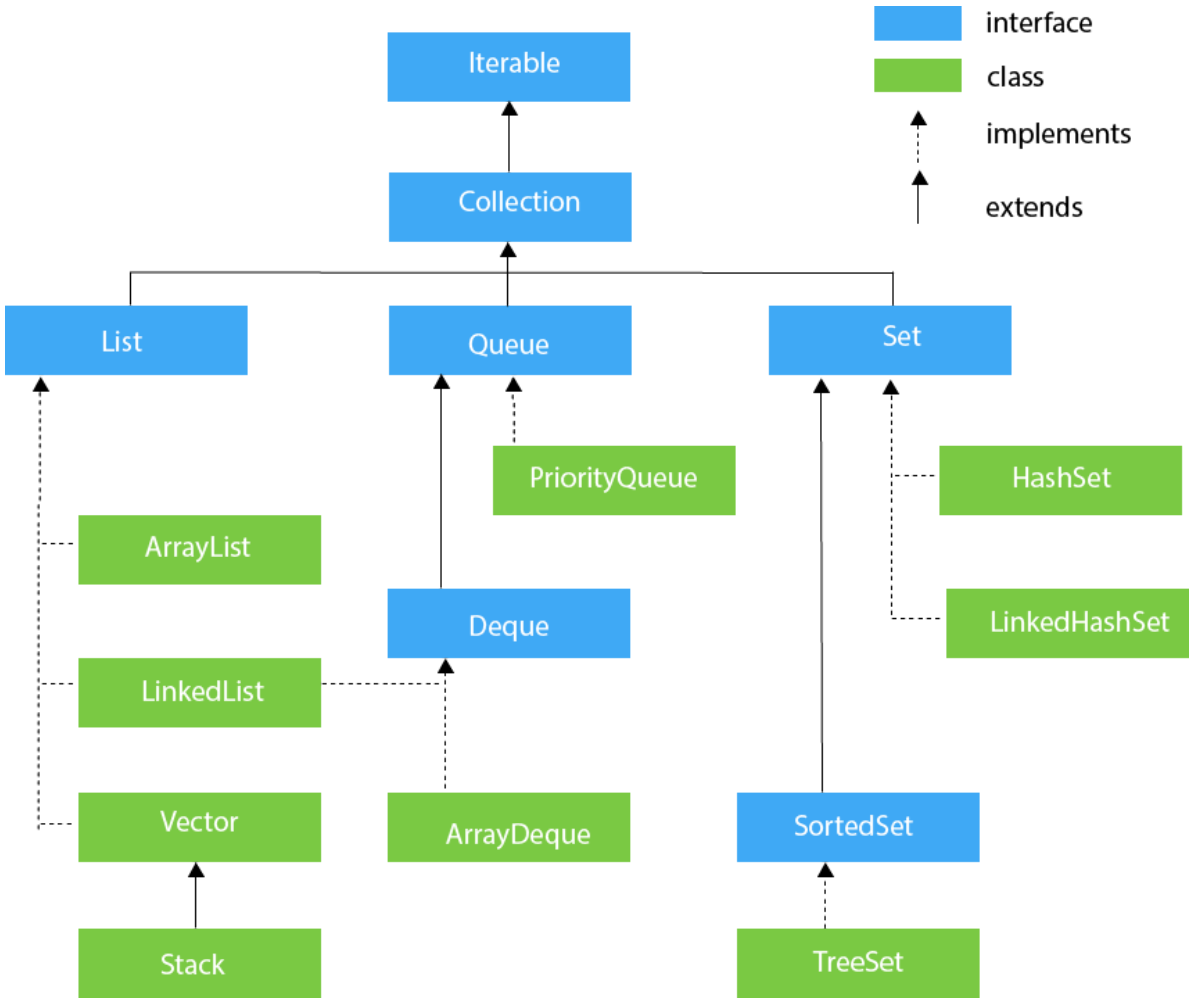


Methods

- `first(): E`
- `last(): E`
- `headSet(E end): SortedSet<E>`
- `subSet(E start, E end): SortedSet<E>`
- `tailSet(E start): SortedSet<E>`

Interface `NavigableSet<E>`

Collection Hierarchy




```
NavigableSet<E> extends SortedSet<E>
```

Methods

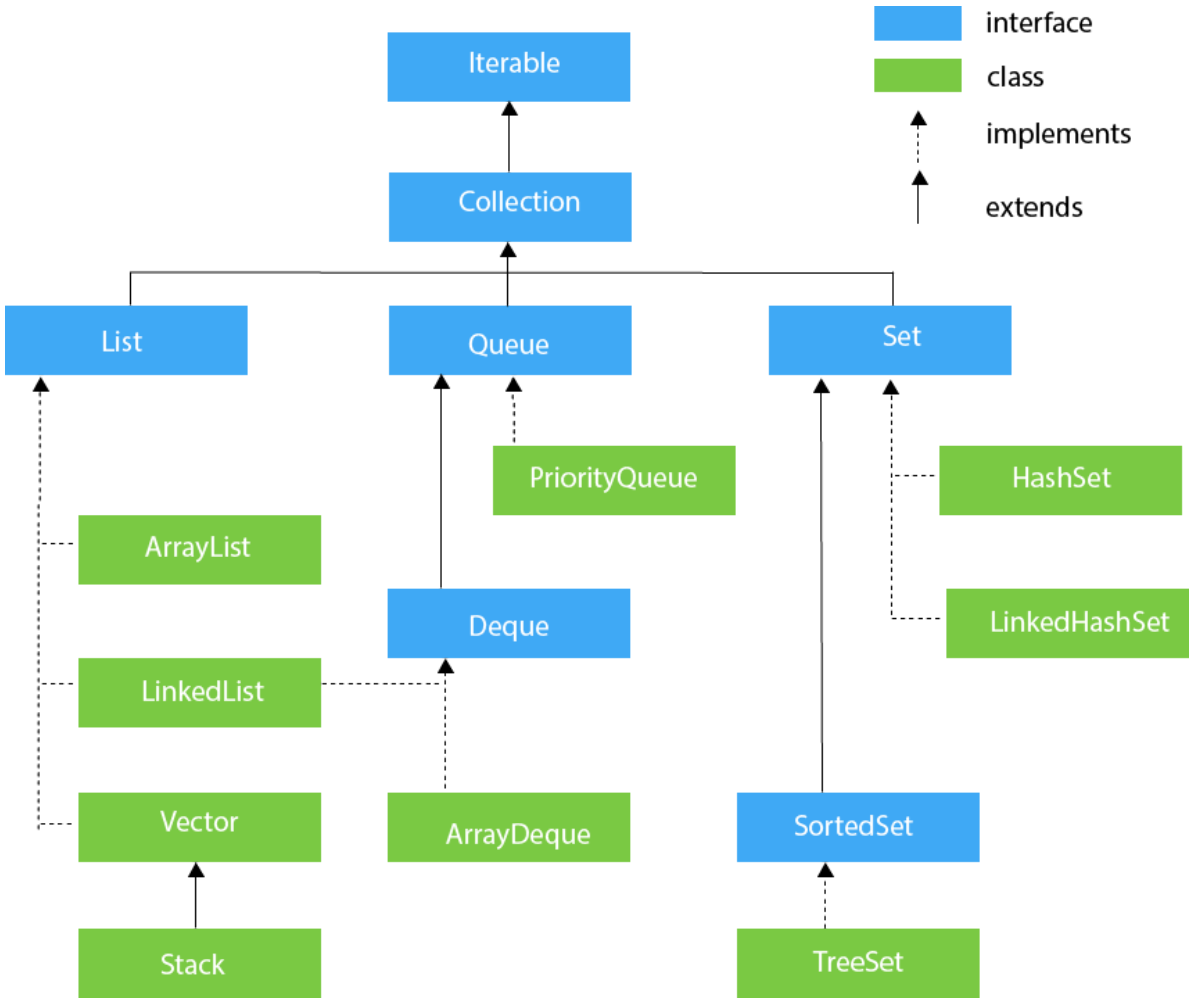
- `ceiling(E obj): E`
- `floor(E obj): E`
- `higher(E obj): E`
- `lower(E obj): E`
- `pollFirst(): E`
- `pollLast(): E`

Methods

- `descendingSet(): NavigableSet<E>`
- `headSet(E upperBound, boolean incl): NavigableSet<E>`
- `tailSet(E lowerBound, boolean incl): NavigableSet<E>`
- `subSet(E lowerBound, boolean lowerIncl, E upperBound, boolean highIncl): NavigableSet<E>`

Class `TreeSet<E>`

Collection Hierarchy



Class `TreeSet<E>`

`TreeSet<E>` - упорядоченное множество элементы которого отсортированы в порядке возрастания.

Constructors

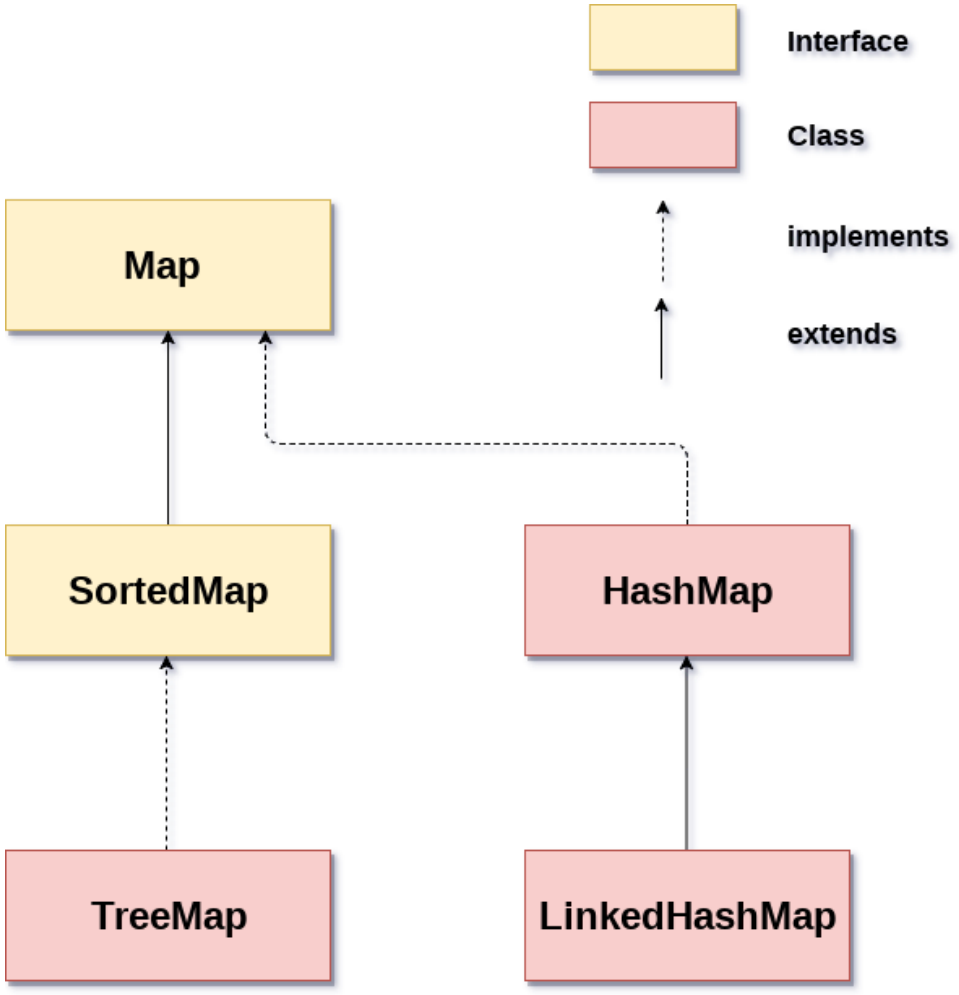
- `TreeSet()`
- `TreeSet(Collection<? extends E> col)`
- `TreeSet(SortedSet <E> set)`
- `TreeSet(Comparator<? super E> comparator)`

Example

```
import java.util.Iterator;
import java.util.TreeSet;

class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<String> al = new TreeSet<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");
        Iterator<String> itr = al.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Interface `Map<K, V>`

Map Hierarchy



Interface `Map<K, V>`

- Управляет парами ключ/значение.
- `Map<K, V>` не может содержать повторяющихся ключей, каждому из которых соответствует не более одного значения.

Methods

- `clear(): void`
- `containsKey(Object k): boolean`
- `containsValue(Object v): boolean`
- `entrySet(): Set<Map.Entry<K, V>>`
- `equals(Object obj): boolean`
- `isEmpty: boolean`
- `get(Object k): V`
- `getOrDefault(Object k, V defaultValue): V`

Methods

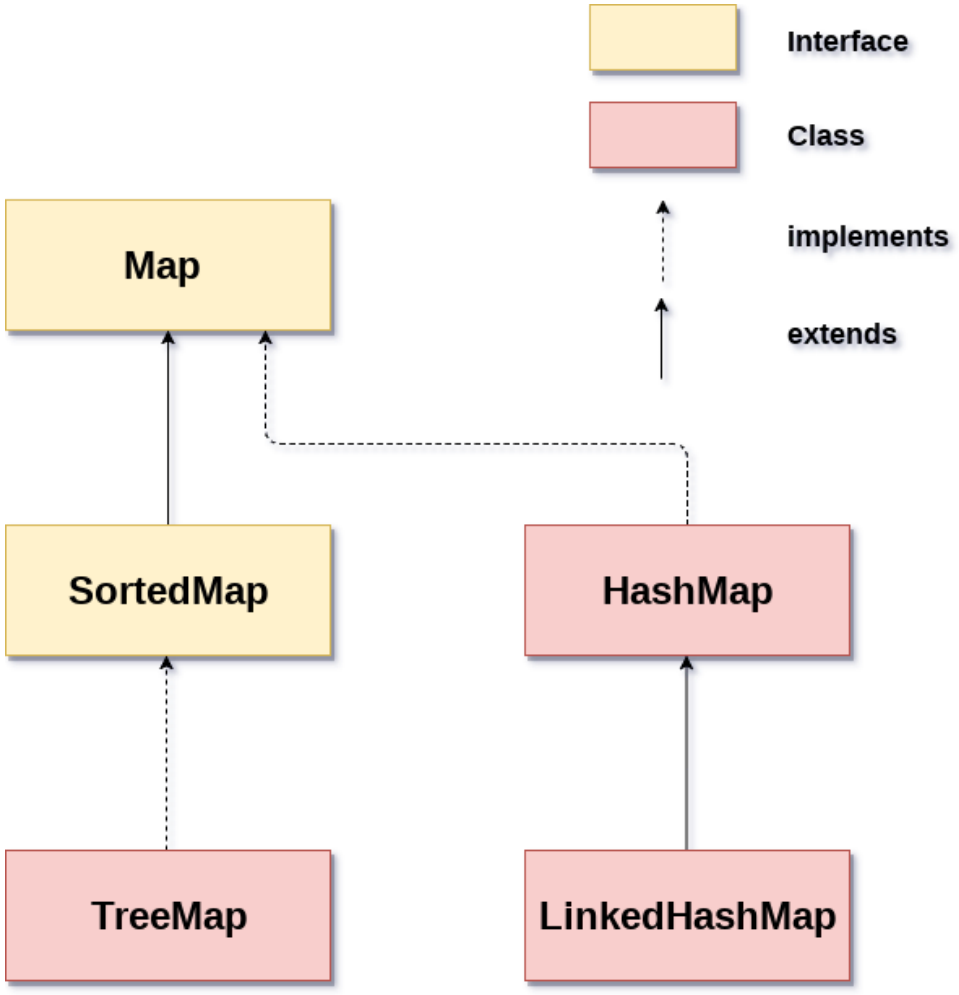
- `put(K k, V v): V`
- `putIfAbsent(K k, V v): V`
- `keySet(): Set<K>`
- `values(): Collection<V>`
- `putAll(Map<? extends K, ? extends V> map): void`
- `remove(Object k): V`
- `size(): int`

Interface `Map.Entry<K, V>`

- `equals(Object obj): boolean`
- `getKey(): K`
- `getValue(): V`
- `keySet(): Set<K>`
- `setValue(V v): V`
- `hashCode(): int`

Class `HashMap<K, V>`

Map Hierarchy



Class `HashMap<K, V>`

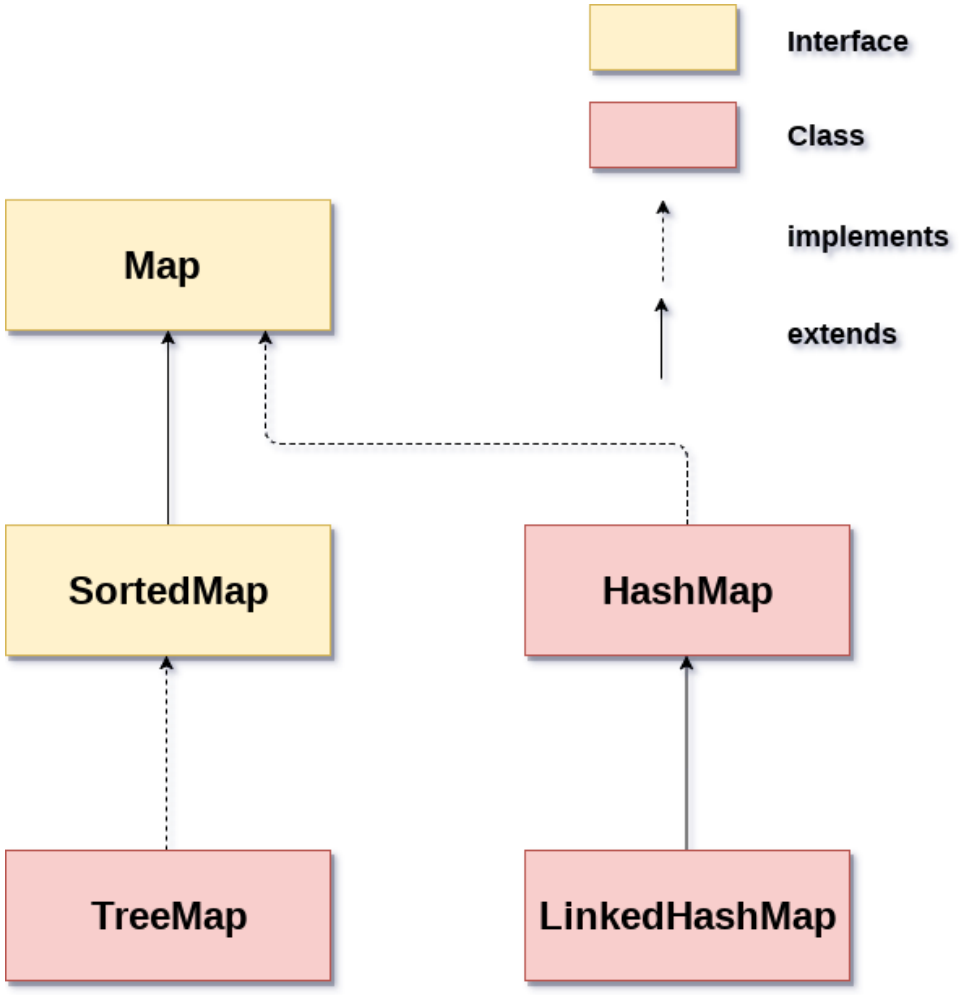
- `HashMap<K, V>` хранит ключи в хеш-таблице, из-за чего имеет наиболее высокую производительность, но не гарантирует порядок элементов.
- Может содержать как `null`-ключи, так и `null`-значения;

Example

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> humans = new HashMap<Integer,
        humans.put(100, "Amit");
        humans.put(101, "Vijay");
        humans.put(102, "Rahul");
        for (Map.Entry m : humans.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue())
        }
    }
}
```

Class `LinkedHashMap<K, V>`

Map Hierarchy

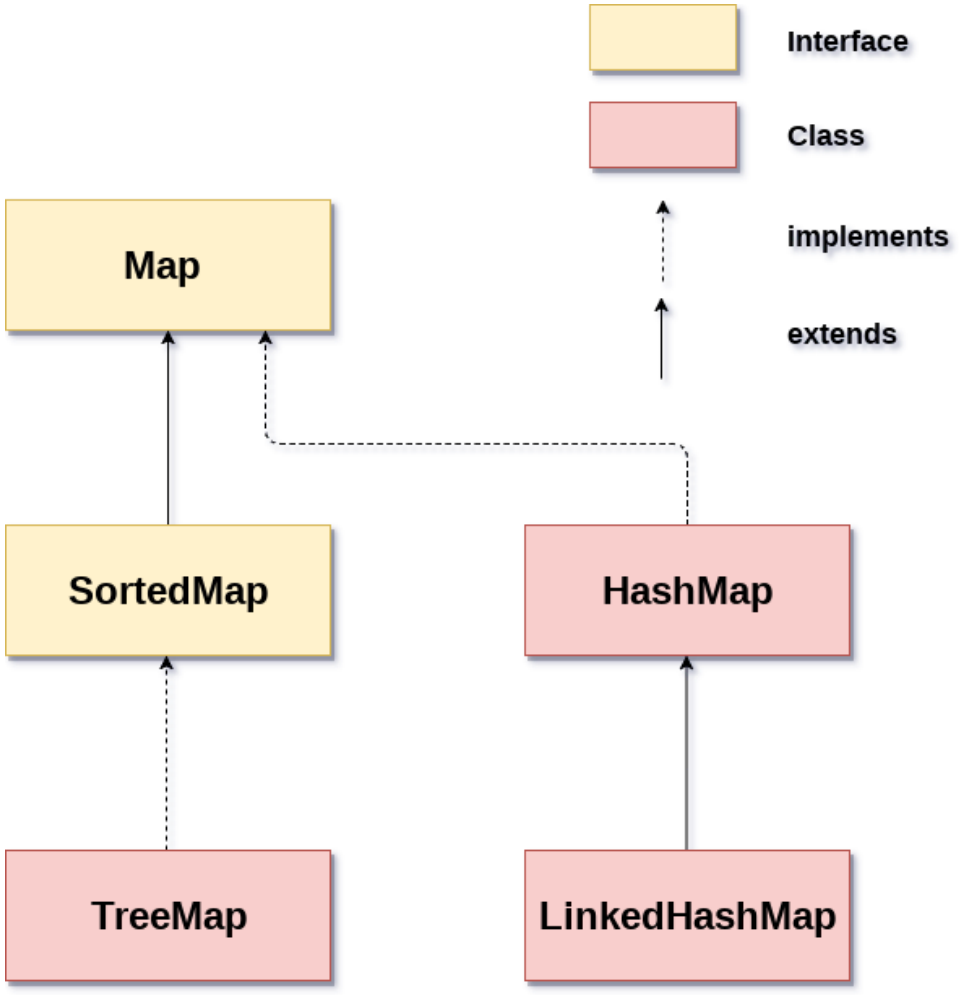


Class `LinkedHashMap<K, V>`

- `LinkedHashMap<K, V>` отличается от `HashMap<K, V>` тем, что хранит ключи в порядке их вставки в `Map<K, V>`.
- Эта реализация `Map<K, V>` лишь немного медленнее `HashMap<K, V>`.
- Может содержать как `null`-ключи, так и `null`-значения.

Interface `SortedMap<K, V>`

Map Hierarchy

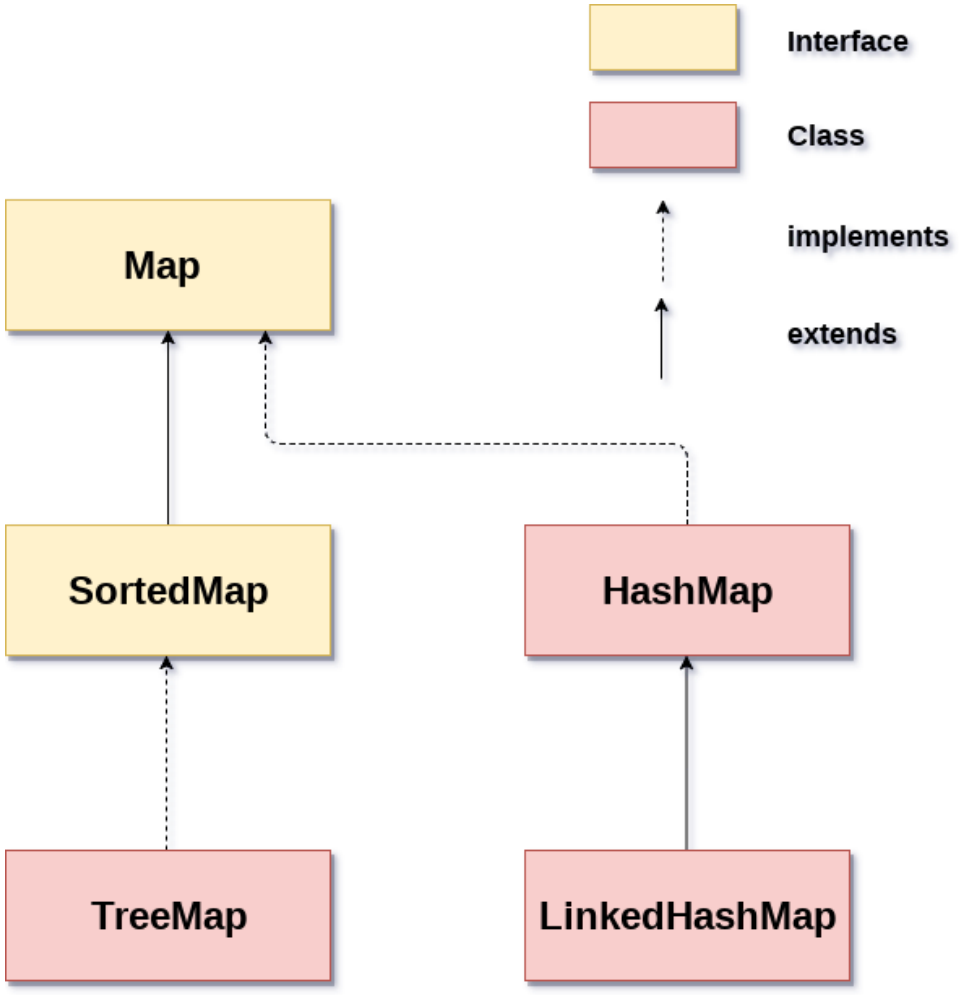


Methods

- `firstKey(): K`
- `lastKey(): K`
- `headMap(K end): SortedMap<K, V>`
- `tailMap(K start): SortedMap<K, V>`
- `subMap(K start, K end): SortedMap<K, V>`

Interface `NavigableMap<K, V>`

Map Hierarchy




```
NavigableMap<K, V> extends SortedMap<K, V>
```

Methods

- `ceilingEntry(K obj): Map.Entry<K, V>`
- `floorEntry(K obj): Map.Entry<K, V>`
- `higherEntry(): Map.Entry<K, V>`
- `lowerEntry(): Map.Entry<K, V>`
- `firstEntry(): Map.Entry<K, V>`
- `lastEntry(): Map.Entry<K, V>`
- `pollFirstEntry(): Map.Entry<K, V>`
- `pollLastEntry(): Map.Entry<K, V>`

Methods

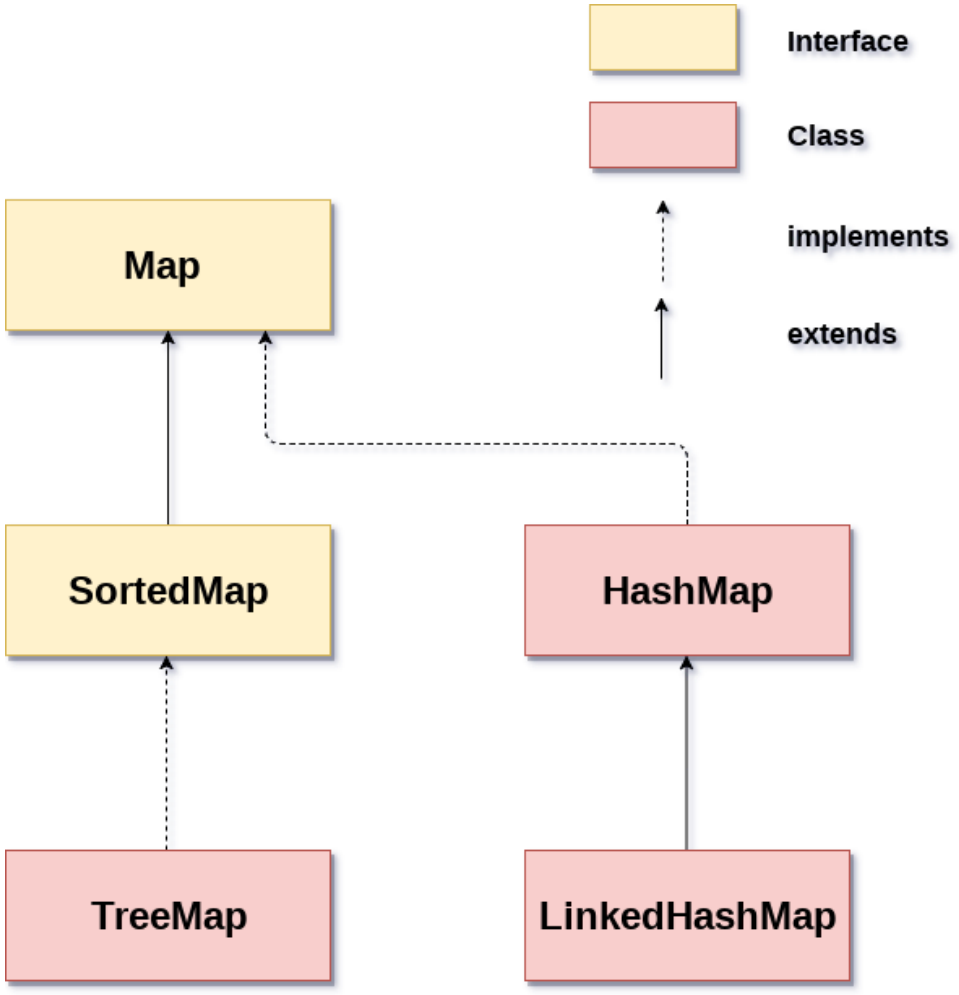
- `ceilingKey(K obj): K`
- `floorKey(K obj): K`
- `lowerKey(K obj): K`
- `higherKey(K obj): K`

Methods

- `descendingKeySet(): NavigableSet<K>`
- `descendingMap(): NavigableMap<K, V>`
- `navigableKeySet(): NavigableSet<K>`
- `headMap(K upperBound, boolean incl): NavigableMap<K, V>`
- `tailMap(K lowerBound, boolean incl): NavigableMap<K, V>`
- `subMap(K lowerBound, boolean lowIncl, K upperBound, boolean highIncl): NavigableMap<K, V>`

Class `TreeMap<K, V>`

Map Hierarchy



Class `TreeMap<K, V>`

- `TreeMap<K, V>` хранит ключи в отсортированном порядке, из-за чего работает существенно медленнее, чем `HashMap<K, V>`.
- Не может содержать `null`-ключи, но может содержать `null`-значения.
- Сортироваться элементы будут либо в зависимости от реализации интерфейса `Comparable`, либо используя объект `Comparator`, который необходимо передать в конструктор `TreeMap<K, V>`;

Constructors

- `TreeMap()`
- `TreeMap(Map<K, ? extends V> map)`
- `TreeMap(SortedMap<K, ? extends V> smap)`
- `TreeMap(Comparator<? super K> comparator)`

Example

```
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<Integer, String> humans = new TreeMap<Integer,
        humans.put(100, "Amit");
        humans.put(102, "Ravi");
        humans.put(101, "Vijay");
        humans.put(103, "Rahul");
        for (Map.Entry m : humans.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue())
        }
    }
}
```

Создание `unmodified` коллекций

Static method `of(…): E`

```
public class Program {
    public static void main(String[] args) {
        List<String> unmodifiedStrings =
                List.of("one", "two", "three");
        System.out.println(unmodifiedStrings);

        List<Integer> unmodifiedInts =
                List.of(1,2,3);
        System.out.println(unmodifiedInts);

        Map<Integer, String> integerStringMap =
                Map.of(1, "one", 2, "two", 3, "three");
        System.out.println(integerStringMap);
    }
}
```

Total

Total

Collection выбирается под задачу.

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))