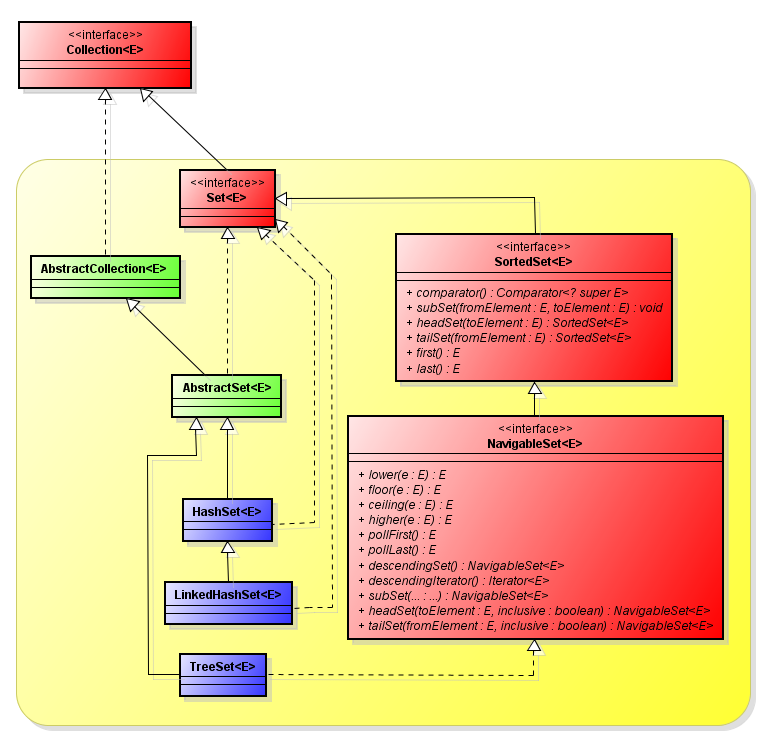
Реализации интерфейса Set



**HashSet**- коллекция, не позволяющая хранить одинаковые объекты(как и любой Set).  **HashSet**инкапсулирует в себе объект **HashMap**(то-есть использует для хранения хэш-таблицу).  
Как большинство читателей, вероятно, знают, хеш-таблица хранит информацию, используя так называемый механизм хеширования, в котором содержимое ключа используется для определения уникального значения, называемого хеш-кодом. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Преобразование ключа в хеш-код выполняется автоматически — вы никогда не увидите самого хеш-кода. Также ваш код не может напрямую индексировать хеш-таблицу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), **remove()** и **size()** , даже для больших наборов.

*Если Вы хотите использовать***HashSet***для хранения объектов СВОИХ классов, то вы ДОЛЖНЫ переопределить методы hashCode() и equals()*, иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод hashCode() класса Object (который скорее-всего вернет разный хэш-код для ваших объектов).  
Важно отметить, что класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов. Если вам нужны сортированные наборы, то лучшим выбором может быть другой тип коллекций, такой как класс TreeSet.

**LinkedHashSet**-  поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор. То есть, когда идет перебор объекта класса LinkedHashSet с применением итератора, элементы извлекаются в том порядке, в каком они были добавлены.

**TreeSet**- коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. TreeSet инкапсулирует в себе TreeMap, который в свою очередь использует сбалансированное бинарное красно-черное дерево для хранения элементов. TreeSet хорош тем, что для операций add, remove и contains потребуется гарантированное время log(n).

|  |  |
| --- | --- |
| **Методы Set** | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Интерфейс **Set** расширяет интерфейс Collection и представляет набор уникальных элементов. Set не добавляет новых методов, только вносит изменения унаследованные. В частности, метод **add()** добавляет элемент в коллекцию и возвращает true, если в коллекции еще нет такого элемента.

Обобщенный класс **HashSet** представляет хеш-таблицу. Он наследует свой функционал от класса AbstractSet, а также реализует интерфейс Set.

Хеш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хеш-код. Данный ключ позволяет уникально идентифицировать объект в таблице.

Для создания объекта HashSet можно воспользоваться одним из следующих конструкторов:

* HashSet(): создает пустой список
* HashSet(Collection<? extends E> col): создает хеш-таблицу, в которую добавляет все элементы коллекции col
* HashSet(int capacity): параметр capacity указывает начальную емкость таблицы, которая по умолчанию равна 16
* HashSet(int capacity, float koef): параметр koef или коэффициент заполнения, значение которого должно быть в пределах от 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами прежде чем произойдет ее расширение. Например, коэффициент 0.75 указывает, что при заполнении емкости на 3/4 произойдет ее расширение.

Класс HashSet не добавляет новых методов, реализуя лишь те, что объявлены в родительских классах и применяемых интерфейсах:

import java.util.HashSet;

public class Program{

    public static void main(String[] args) {

        HashSet<String> states = new HashSet<String>();

        // добавим в список ряд элементов

        states.add("Germany");

        states.add("France");

        states.add("Italy");

        // пытаемся добавить элемент, который уже есть в коллекции

        boolean isAdded = states.add("Germany");

        System.out.println(isAdded);    // false

        System.out.printf("Set contains %d elements \n", states.size());    // 3

        for(String state : states){

            System.out.println(state);

        }

        // удаление элемента

        states.remove("Germany");

        // хеш-таблица объектов Person

        HashSet<Person> people = new HashSet<Person>();

        people.add(new Person("Mike"));

        people.add(new Person("Tom"));

        people.add(new Person("Nick"));

        for(Person p : people){

            System.out.println(p.getName());

        }

    }

}

class Person{

    private String name;

    public Person(String value){

        name=value;

    }

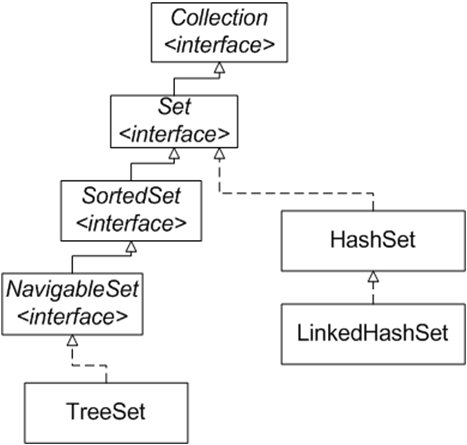
    String getName(){return name;}

}

**Интерфейс Set и Реализации**

**1. Интерфейс *Set***

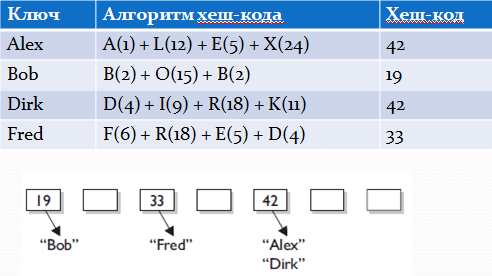
Интерфейс *Set* определяет множество (набор). Он расширяет *Collection*и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод *add()*возвращает *false*, если делается попытка добавить дублированный элемент в набор. Он не определяет никаких собственных дополнительных методов. Интерфейс *Set*заботится об уникальности хранимых объектов, уникальность определятся реализацией метода *equals().*

**

**2. Класс *HashSet***

Класс *HashSet* реализует интерфейс *Set*. Он создает коллекцию, которая используется для хранения хеш-таблиц. Элементы таблицы хранятся в виде пар ключ-значение. Ключ определяет  ячейку для хранения значения. Содержимое ключа служит для определения однозначного значения, называемого хеш-кодом. Мы можем считать, что хеш-код это ID объекта, хотя он не должен быть уникальным. Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с ключом.

**Рисунок 1. Использование метода *hashCode()***



#### Правила написания методов hashCode() и equals():

1. Для одного и того же объекта, хеш-код всегда будет одинаковым.
2. Если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот).
3. Если хеш-коды равны, то входные объекты не всегда равны.
4. Если хеш-коды разные, то и объекты гарантированно разные.

Выгода от хеширования состоит в том, что оно обеспечивает постоянство время выполнения операций add(), contains(), remove() и size(), даже для больших наборов. Класс HashSet не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств.

#### Пример 1. Использование класса HashSet

import java.util.HashSet;

import java.util.Set;

public class HashSetDemo {

public static void main(String[] args) {

Set<String> hashSet = new HashSet<>();

hashSet.add("Бета");

hashSet.add("Aльфa");

hashSet.add("Этa");

hashSet.add("Гaммa");

hashSet.add("Эпсилон");

hashSet.add("Oмeгa");

System.out.println(hashSet);

}

}

## 3. Класс LinkedHashSet

Класс LinkedHashSet расширяет HashSet, не добавляя никаких новых методов. LinkedHashSet поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор.

#### Пример 2. Использование класса LinkedHashSet

import java.util.LinkedHashSet;

import java.util.Set;

public class LinkedHashSetDemo {

public static void main(String[] args) {

Set<String> linkedHashSet = new LinkedHashSet<>();

linkedHashSet.add("Бета");

linkedHashSet.add("Aльфa");

linkedHashSet.add("Этa");

linkedHashSet.add("Гaммa");

linkedHashSet.add("Эпсилон");

linkedHashSet.add("Oмeгa");

System.out.println(linkedHashSet);

}

}

## 4. Интерфейс SortedSet

Интерфейс SortedSet из пакета java.util, расширяющий интерфейс Set, описывает упорядоченное множество, отсортированное по естественному порядку возрастания его элементов или по порядку, заданному реализацией интерфейса Comparator.

#### Методы интерфейса SortedSet:

1. **Comparator<? super E> comparator()** - возвращает компаратор сортированного множества. Если для множества применяется естественный порядок сортировки, возвращается null.
2. ***E first()*** - возвращает первый элемент вызывающего сортированного множества.
3. **E last()** - возвращает последний элемент вызывающего сортированного множества.
4. ***SortedSet headSet(E toElement)*** - возвращает SortedSet, содержащий элементы из вызывающего множества, которые предшествуют end.
5. ***SortedSet subSet(E fromElement, E toElement)*** - возвращает SortedSet, содержащий элементы из вызывающего множества, находящиеся между start и end-1.
6. ***SortedSet tailSet(E fromElement)*** - возвращает SortedSet, содержащий элементы из вызывающего множества, которые следуют за end.

#### Пример 3. Использование методов subSet(), headSet(), tailSet()

import java.util.SortedSet;

import java.util.TreeSet;

public class TreeSetDemo2 {

public static void main(String[] args) {

SortedSet<String> treeSet = new TreeSet<>();

treeSet.add("Бета");

treeSet.add("Aльфa");

treeSet.add("Этa");

treeSet.add("Гaммa");

treeSet.add("Эпсилон");

treeSet.add("Омeгa");

System.out.println(treeSet);

SortedSet subSet = treeSet.subSet("Бета", "Омeгa");

System.out.println("SubSet: " + subSet);

System.out.println("HeadSet: " + treeSet.headSet("Гaммa"));

System.out.println("TailSet: " + treeSet.tailSet("Гaммa"));

}

}

## 5. Класс TreeSet

TreeSet<E> – реализует интерфейс NavigableSet<E>, который поддерживает элементы в отсортированном по возрастанию порядке. Объекты сохраняются в отсортированном порядке по нарастающей. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

#### Пример 4. Использование класса TreeSet

import java.util.SortedSet;

import java.util.TreeSet;

public class TreeSetDemo1 {

public static void main(String args[]) {

SortedSet<String> treeSet = new TreeSet<>();

treeSet.add("Бета");

treeSet.add("Aльфa");

treeSet.add("Этa");

treeSet.add("Гaммa");

treeSet.add("Эпсилон");

treeSet.add("Омeгa");

System.out.println(treeSet);

}

}

#### 

#### Конструкторы класса *****TreeSet:*****

* TreeSet()
* TreeSet(Collection<? extends Е> сollection)
* TreeSet(Comparator<? super Е> соmрarator)
* TreeSet(SortedSet<E> sortedSet)

## 6. Сравнение объектов

Существует два способа сравнения объектов:

* С помощью интерфейса Comparable.
* С помощью интерфейса Comparator.

### 6.1. Интерфейс Comparable<T>

Для того, чтобы объекты можно было сравнить и сортировать, они должны реализовать интерфейс Comparable<T>. Интерфейс Comparable<T> содержит один единственный метод int compareTo(T item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвращает ноль, значит, оба объекта равны.

#### Пример 5. Использования интерфейса Comparable<T>

public class Person implements Comparable<Person> {

private String firstName;

private String lastName;

private int age;

public Person(String firstName, String lastName, int age) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

@Override

public int compareTo(Person anotherPerson) {

int anotherPersonAge = anotherPerson.getAge();

return this.age - anotherPersonAge;

}

@Override

public String toString() {

return "Person{" +

"firstName='" + firstName + '\'' +

", lastName='" + lastName + '\'' +

", age=" + age +

'}';

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Person person = (Person) o;

if (getAge() != person.getAge()) return false;

if (getFirstName() != null ? !getFirstName().equals(person.getFirstName()) : person.getFirstName() != null)

return false;

return getLastName() != null ? getLastName().equals(person.getLastName()) : person.getLastName() == null;

}

@Override

public int hashCode() {

int result = getFirstName() != null ? getFirstName().hashCode() : 0;

result = 31 \* result + (getLastName() != null ? getLastName().hashCode() : 0);

result = 31 \* result + getAge();

return result;

}

}

import java.util.SortedSet;

import java.util.TreeSet;

public class ComparePersonDemo {

public static void main(String[] args) {

SortedSet<Person> set = new TreeSet<>();

set.add(new Person("Саша", "Иванов", 36));

set.add(new Person("Маша", "Петрова", 23));

set.add(new Person("Даша", "Сидорова", 34));

set.add(new Person("Вася", "Иванов", 25));

for (Person person : set) {

System.out.println(person);

}

}

}

### 6.2. Интерфейс Comparator<T>

Интерфейс Comparator<T> содержит метод: int compare(T o1, T o2). Метод compare также возвращает числовое значение - если оно отрицательное, то объект o1 предшествует объекту o2, иначе - наоборот. А если метод возвращает ноль, то объекты равны. Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс.

#### Пример 6. Использование интерфейса Comparator<T>

import java.util.Comparator;

public class PersonComparator implements Comparator<Person> {

@Override

public int compare(Person o1, Person o2) {

return o1.getLastName().compareTo(o2.getLastName());

}

}

import java.util.SortedSet;

import java.util.TreeSet;

public class PersonComparatorDemo {

public static void main(String[] args) {

PersonComparator personComparator = new PersonComparator();

SortedSet<Person> set = new TreeSet<>(personComparator);

set.add(new Person("Саша", "Иванов", 36));

set.add(new Person("Маша", "Петрова", 23));

set.add(new Person("Даша", "Сидорова", 34));

set.add(new Person("Вася", "Иванов", 25));

//Обратите внимание - было добавлено 4 элемента, но распечатано 3

for (Person person : set) {

System.out.println(person);

}

}

}

## 7. Интерфейс NavigableSet

Интерфейс NavigableSet появился в Java SE 6. Он расширяет SortedSet и добавляет методы для более удобного поиска по коллекции:

1. **Е ceiling(E obj)** - ищет в наборе наименьший элемент е, для которого истинно е >= obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.
2. **Е floor(Е obj)** - ищет в наборе наибольший элемент е, для которого истинно е <= obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.
3. **Е higher(Е obj)** - ищет в наборе наибольший элемент е, для которого истинно е > obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.
4. **Е lower(Е obj)** - ищет в наборе наименьший элемент е, для которого истинно е < obj. Если такой элемент найден, он возвращается. В противном случае возвращается null.
5. ***NavigableSet headSet(Е upperBound, boolean incl)*** - возвращает NavigableSet, включающий все элементы вызывающего набора, меньшие upperBound. Результирующий набор поддерживается вызывающим набором.
6. **NavigableSet subSet(Е lowerBound, boolean lowlncl, Е upperBound, boolean highIncl)** - возвращает NavigableSet, включающий все элементы вызывающего набора, которые больше lowerBound и меньше upperBound. Если lowlncl равно true, то элемент, равный lowerBound, включается. Если highlncl равно true, также включается элемент, равный upperBound.
7. ***E pollLast()*** - возвращает последний элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наибольшим значением. Возвращает null в случае пустого набора.
8. **Е pollFirst()** - возвращает первый элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наименьшим значением. Возвращает null в случае пустого набора.
9. ***Iterator descendingIterator()*** - возвращает итератор, перемещающийся от большего к меньшему, другими словами, обратный итератор.
10. **NavigableSet descendingSet()** - возвращает NavigableSet, представляющий собой обратную версию вызывающего набора. Результирующий набор поддерживается вызывающим набором.

#### Пример 7. Использования NavigableSet

import java.util.NavigableSet;

import java.util.SortedSet;

import java.util.TreeSet;

public class Ferry {

public static void main(String[] args) {

NavigableSet<Integer> times = new TreeSet<>();

times.add(1205); // add some departure times

times.add(1505);

times.add(1545);

times.add(1830);

times.add(2010);

times.add(2100);

// Java 5 version

SortedSet<Integer> subset = times.headSet(1600);

System.out.println("J5 - last before 4pm is: " + subset.last());

SortedSet<Integer> sub2 = times.tailSet(2000);

System.out.println("J5 - first after 8pm is: " + sub2.first());

// Java 6 version using the new lower() and higher() methods

System.out.println("J6 - last before 4pm is: " + times.lower(1600));

System.out.println("J6 - first after 8pm is: " + times.higher(2000));

}

}