6. Java Collections Framework

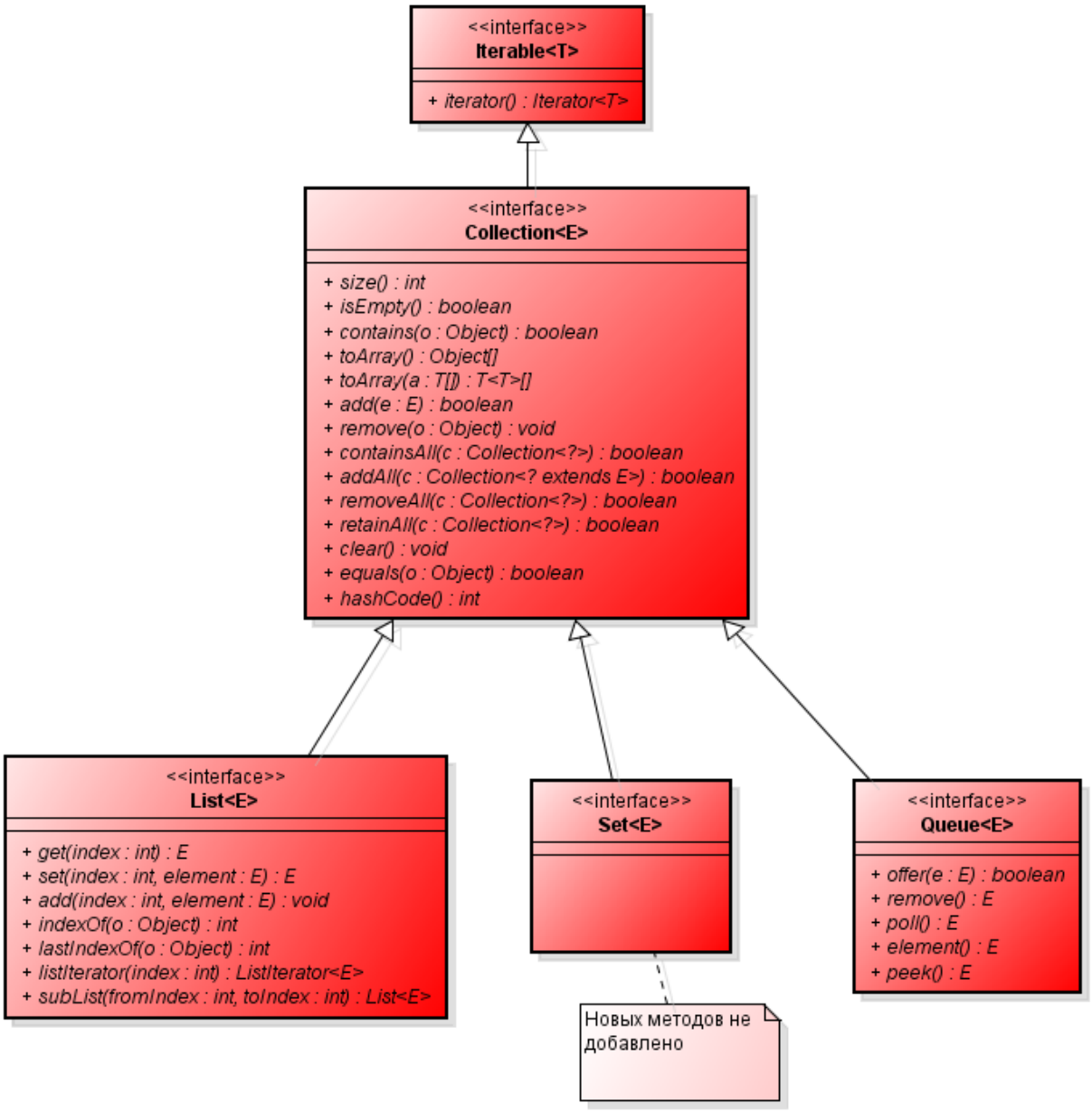
Базовые интерфейсы

В библиотеке коллекций Java существует два базовых интерфейса, реализации которых и представляют совокупность всех классов коллекций:

1. `Collection` - коллекция содержит набор объектов (элементов). Здесь определены основные методы для манипуляции с данными, такие как вставка (`add` , `addAll`), удаление (`remove` , `removeAll` , `clear`), поиск (`contains`)
2. `Map` - описывает коллекцию, состоящую из пар "ключ — значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (map). Такую коллекцию часто называют еще словарем (dictionary) или ассоциативным массивом (associative array). Никак НЕ относится к интерфейсу Collection и является самостоятельным.

Интерфейс Collection

Давайте рассмотрим основные интерфейсы, относящиеся к `Collection`:

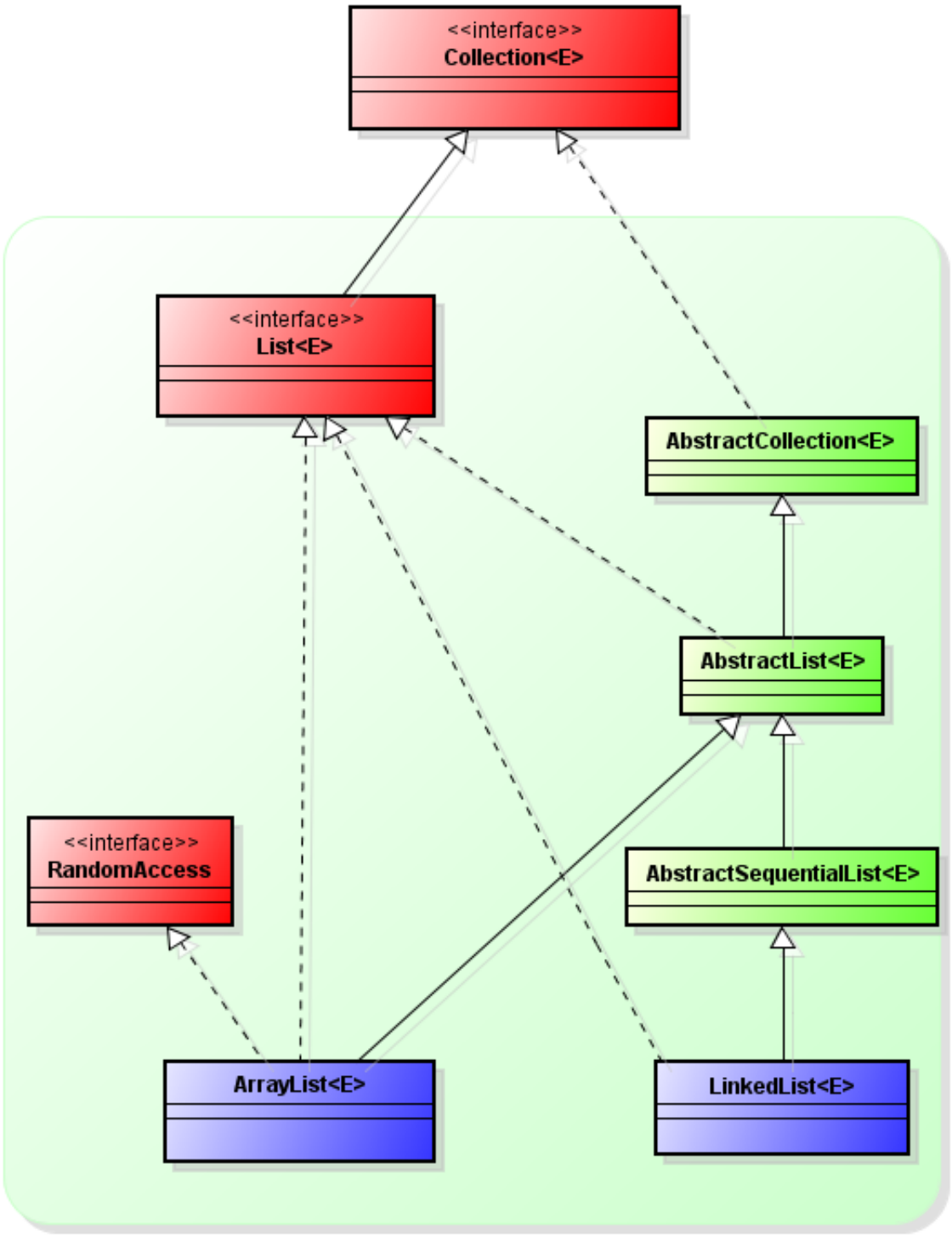


Интерфейс `Collection` расширяет интерфейс `Iterable`, у которого есть только один метод `iterator()`. Это значит что любая коллекция, которая есть наследником `Iterable` должна возвращать итератор.

Как видим с рисунка, интерфейс `Collection` расширяют интерфейсы `List`, `Set` и `Queue`. Давайте рассмотрим, зачем нужен каждый.

1. `List` - Представляет собой неупорядоченную коллекцию, в которой допустимы дублирующие значения. Иногда их называют последовательностями (`sequence`). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.
2. `Set` - описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (`set`).
3. `Queue` - очередь. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Реализации интерфейса List



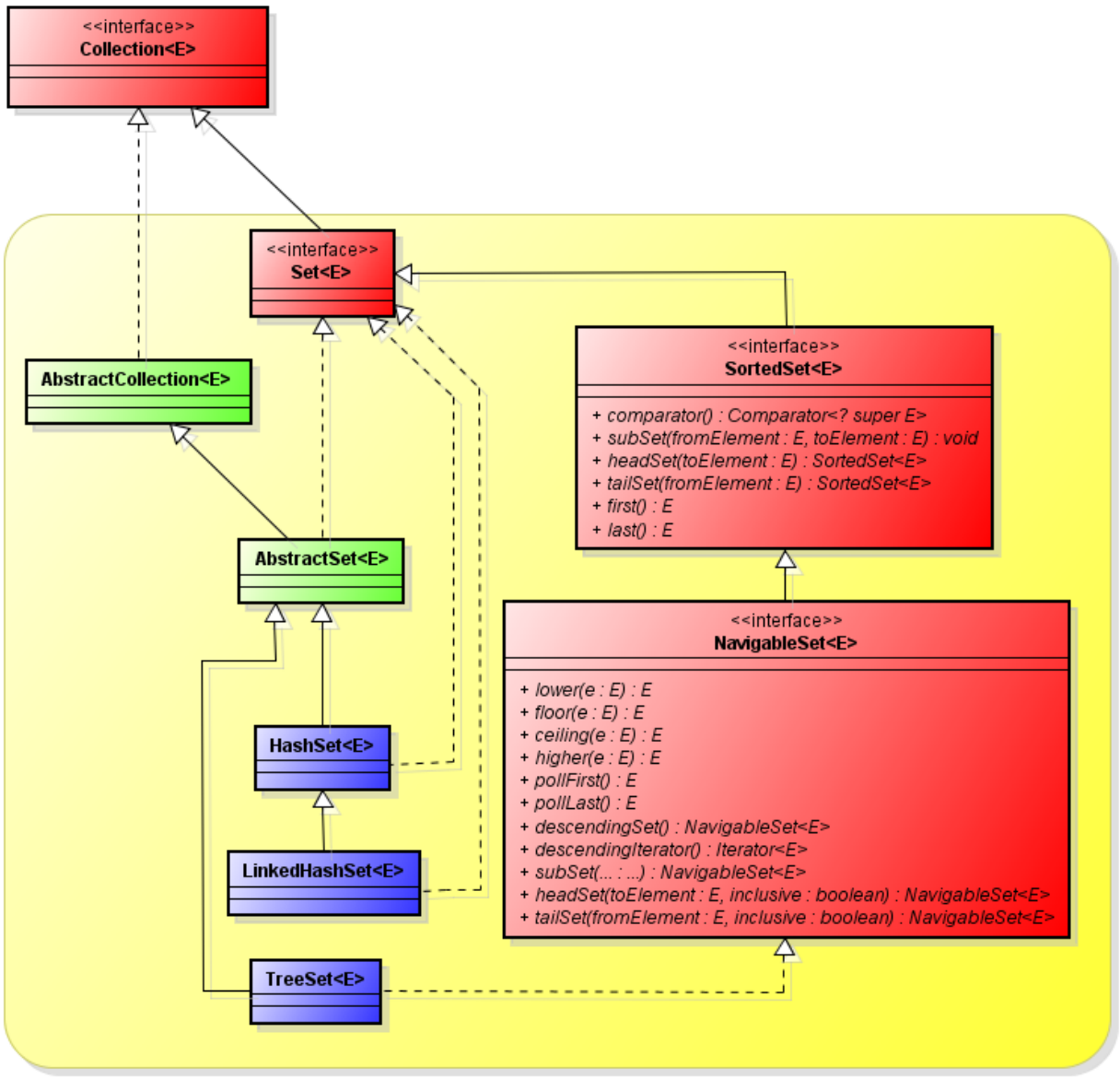
~Красным~ здесь выделены интерфейсы, ~зеленым~ - абстрактные классы, а ~синим~ готовые реализации. Сразу хочу заметить что здесь не вся иерархия, а только основная её часть.

Как видим на рисунке, между интерфейсом и конкретной реализацией коллекции существует несколько абстрактных классов. Это сделано для того, что бы вынести общий функционал в абстрактный класс, таким образом реализовать повторное использование кода.

`ArrayList` - пожалуй самая часто используемая коллекция. `ArrayList` инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.
Так как `ArrayList` использует массив, то время доступа к элементу по индексу минимально (В отличии от `LinkedList`). При удалении произвольного элемента из списка, все элементы находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, `capacity`) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером `(n * 3) / 2 + 1`, в него будут помещены все элементы из старого массива + новый, добавляемый элемент.

`LinkedList` — двусвязный список. Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки («связки») на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, `LinkedList` в абсолютных величинах проигрывает `ArrayList` и по потребляемой памяти и по скорости выполнения операций.

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.

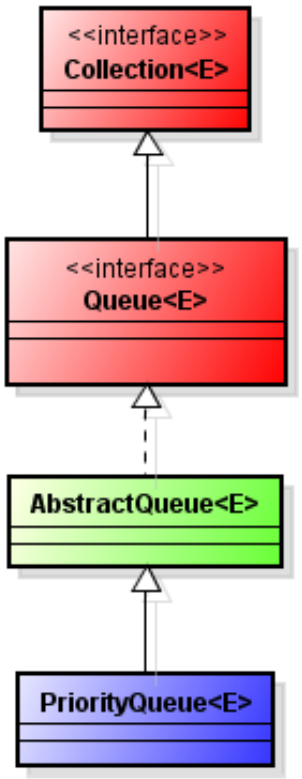
Если Вы хотите использовать `HashSet` для хранения объектов СВОИХ классов, то вы ДОЛЖНЫ переопределить методы `hashCode()` и `equals()`, иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод `hashCode()` класса `Object` (который 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)`.

Реализации интерфейса Queue



`PriorityQueue` — единственная прямая реализация интерфейса `Queue` (не считая `LinkedList`, который больше является списком, чем очередью).
Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс `Comparable`), либо с помощью интерфейса `Comparator`, полученному в конструкторе.

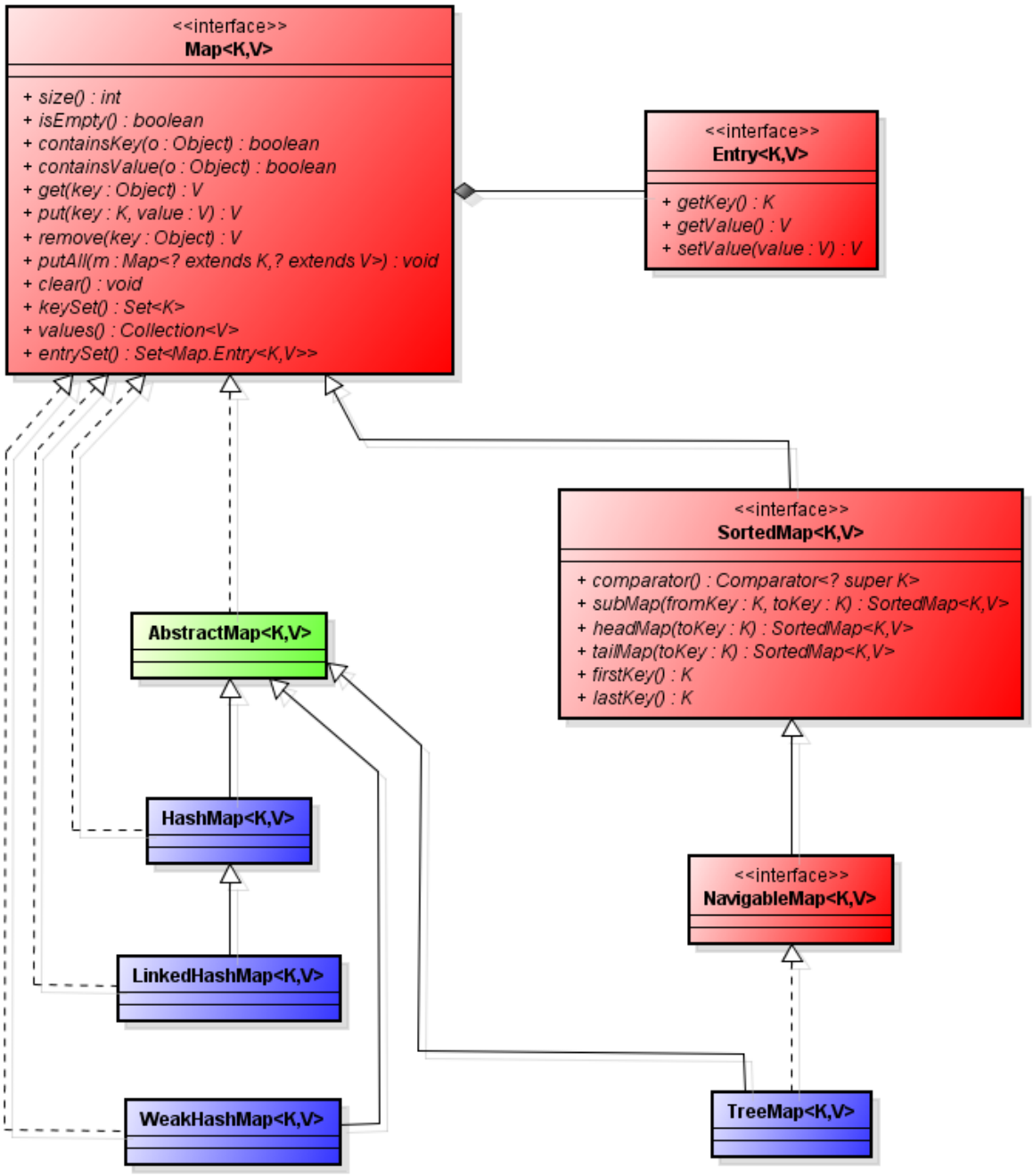
Реализации интерфейса Map

Интерфейс `Map` соотносит уникальные ключи со значениями. Ключ — это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект карты. После того как это значение сохранено, вы можете получить его по ключу. Интерфейс Map — это обобщенный интерфейс, объявленный так, как показано ниже.

```
interface Map<K, V>
```

Здесь `K` указывает тип ключей, а `V` — тип хранимых значений.

Иерархия классов очень похожа на иерархию `Set`'а:



`HashMap` — основан на хэш-таблицах, реализует интерфейс `Map` (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и `null`. Данная реализация не дает гарантий относительно порядка элементов с течением времени.

`LinkedHashMap` - расширяет класс `HashMap`. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать перебор карты в порядке вставки. То есть, когда происходит итерация по коллекционному представлению объекта класса `LinkedHashMap`, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать объект класса `LinkedHashMap`, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ.

`TreeMap` - расширяет класс `AbstractMap` и реализует интерфейс `NavigatebleMap`. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает класс `TreeMap` блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.

`WeakHashMap` - коллекция, использующая слабые ссылки для ключей (а не значений). Слабая ссылка (англ. weak reference) — специфический вид ссылок на динамически создаваемые объекты в системах со сборкой мусора. Отличается от обычных ссылок тем, что не учитывается сборщиком мусора при выявлении объектов, подлежащих удалению. Ссылки, не являющиеся слабыми, также иногда именуют «сильными».

Устаревшие коллекции

Следующие коллекции являются устаревшими, и их использование не рекомендуется, но не запрещается.

1. `Enumeration` — аналог интерфейса `Iterator`.
2. `Vector` — аналог класса `ArrayList`; поддерживает упорядоченный список элементов, хранимых во "внутреннем" массиве.
3. `Stack` — класс, производный от Vector, в который добавлены методы вталкивания (push) и выталкивания (pop) элементов, так что список может трактоваться в терминах, принятых для описания структуры данных стека (stack).
4. `Dictionary` — аналог интерфейса `Map`, хотя представляет собой абстрактный класс, а не интерфейс.
5. `Hashtable` — аналог `HashMap`.

Все методы `Hashtable`, `Stack`, `Vector` являются ~синхронизированными~, что делает их менее эффективными в одно поточных приложениях.

Синхронизированные коллекции

Получить синхронизированные объекты коллекций можно с помощью статических методов `synchronizedMap` и `synchronizedList` класса `Collections`.

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
```

Синхронизированные обрамления коллекций `synchronizedMap` и `synchronizedList` иногда называют условно потокобезопасными - все операции в отдельности потокобезопасны, но последовательности операций, где управляющий поток зависит от результатов предыдущих операций, могут быть причиной конкуренции за данные.

Условная безопасность потоков, обеспечиваемая `synchronizedList` и `synchronizedMap` представляет скрытую угрозу - разработчики полагают, что, раз эти коллекции синхронизированы, значит, они полностью потокобезопасны, и пренебрегают должной синхронизацией составных операций. В результате, хотя эти программы и работают при лёгкой нагрузке, но при серьёзной нагрузке они могут начать выкидывать `NullPointerException` или `ConcurrentModificationException`.

Кроме того всегда существует возможность "классической" синхронизации с помощью блока `synchronized`.

Итератор

Одним из ключевых методов интерфейса `Collection` является метод `Iterator<E> iterator()`. Он возвращает итератор - то есть объект, реализующий интерфейс `Iterator`.

Интерфейс `Iterator` имеет следующее определение:

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
    void remove();
}
```

Реализация интерфейса предполагает, что с помощью вызова метода `next()` можно получить следующий элемент. С помощью метода `hasNext()` можно узнать, есть ли следующий элемент, и не достигнут ли конец коллекции. И если элементы еще имеются, то `hasNext()` вернет значение `true`. Метод `hasNext()` следует вызывать перед методом `next()`, так как при достижении конца коллекции метод `next()` выбрасывает исключение `NoSuchElementException`. И метод `remove()` удаляет текущий элемент, который был получен последним вызовом `next()`.

Используем итератор для перебора коллекции `ArrayList`:

```java
import java.util.*;

public class CollectionApp {
    public static void main(String[] args) {

        ArrayList<String> states = new ArrayList<String>();
        states.add("Германия");
        states.add("Франция");
        states.add("Италия");
        states.add("Испания");

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