нетология

Коллекции HashMap и HashSet

Юрий
Пеньков

Юрий Пеньков

Senior Java Developer в CIT RT

Юрий Пеньков

План занятия

1. Хеш-таблица
2. Реализация коллекции `HashMap`
3. Реализация коллекции `HashSet`

Хеш-таблица

Задача

Дано

Мы пишем метод для получения списка водительских удостоверений, выданных в любом конкретном городе. Каждая строка прочитанных данных о водительском удостоверении содержит следующий набор полей:

- Идентификатор водительского удостоверения;
- Идентификатор водителя;
- Дата выдачи удостоверения;
- Срок действия;
- Набор категорий.

Вопрос

В какой структуре данных можно хранить такие данные при условиях:

- эти данные являются объектами, полученными из базы данных или прочитанными из файла;
- нам требуется находить каждого водителя по его номеру удостоверения.

Ответ

Зависит от дальнейшего способа использования этих данных, возможные варианты:

- массивы,
- списки.

Можно хранить объекты в массивах или списках. Такой способ возможен, но для поиска нужного элемента в таких коллекциях, например, по номеру водительского удостоверения, нужно будет перебрать все элементы этой коллекции. Это будет работать медленно уже на коллекциях размером в несколько тысяч элементов. Сложность такого поиска будет равна O(n).

Для ускорения поиска объектов по идентификатору, были разработаны коллекции на основе **хеш-таблиц**. Такие таблицы позволяют ускорить поиск элементов в них до сложности O(1).

Хеш-таблица

Хеш-таблица — это структура данных, представляющая собой ассоциативный массив, то есть структура, хранящая пары ключ-значение.

В хеш-таблицы можно хранить пары вида **идентификатор — объект, название — объект, номер — объект** и т.д.

Поддерживает стандартные операции:

- Добавление пары;
- Поиск объекта по ключу;
- Удаление по ключу.

Любая хеш-таблица, как следует из ее названия, основана на хешировании.

Что такое хеширование?

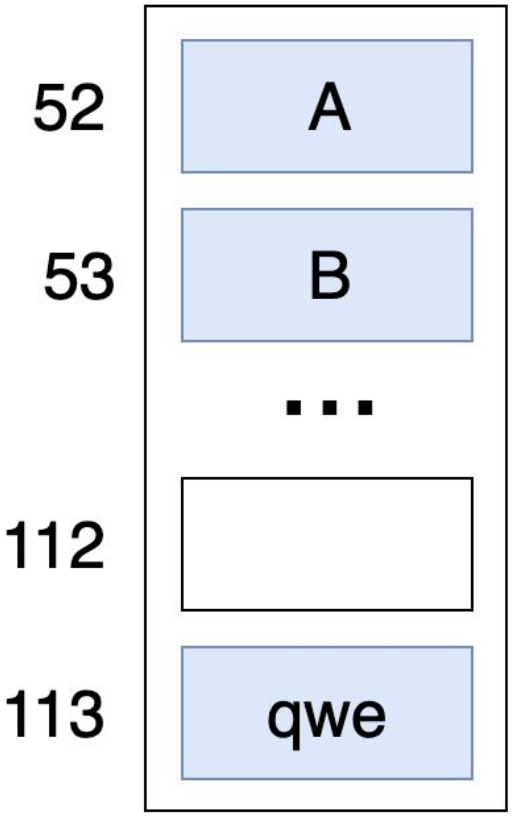
Хеширование — это превращение входных данных произвольной длины в выходную строку фиксированной длины или число.

Пример:

```
"A".hashCode()    // 65
"B".hashCode()    // 66
"qwe".hashCode()  // 112383
```

Что такое хеширование?

Хеш-таблица использует этот код, чтобы найти элемент без перебора всех ключей. В простейшем случае по хеш-коду определяется индекс в массиве (номер ячейки), по которому находится элемент.



Вопрос

Может ли для разных строк или любых других объектов вычисляться один и тот же хеш-код?

Вопрос

Может ли для разных строк или любых других объектов вычисляться один и тот же хеш-код?

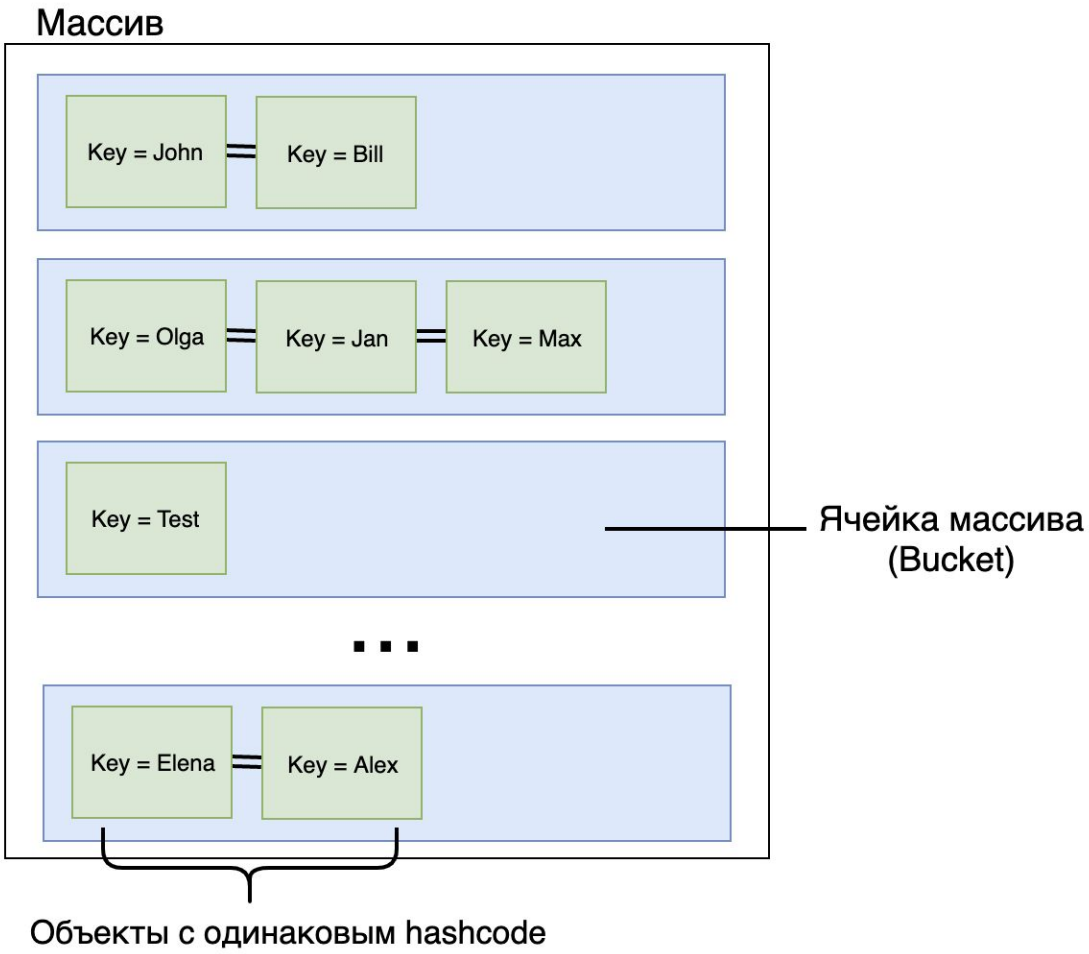
Ответ

Да, такое явление называется **коллизией**.

Для разрешения коллизий используются разные методы.

В Java, **если по одному индексу**, определенному по хешкоду, **находятся несколько элементов, то нужный находится с помощью вызовов** `equals` на каждом из элементов, находящихся в этой ячейке массива.

Пример хеш-таблицы



Реализация коллекции HashMap

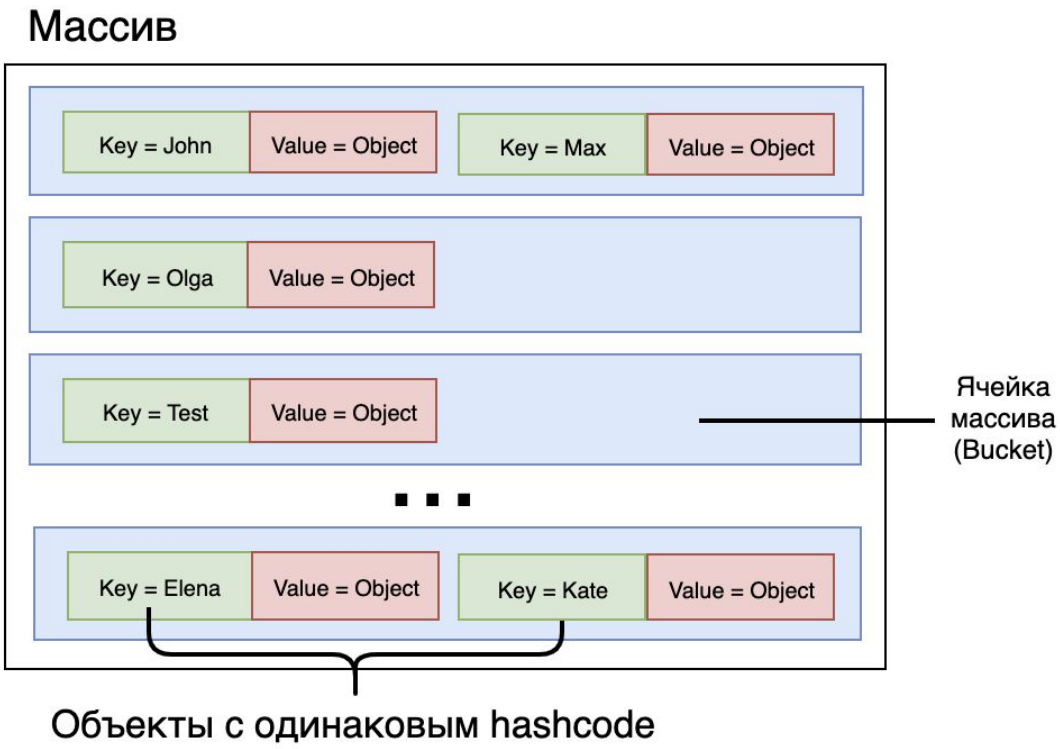
Коллекция HashMap

Ответ на вопрос, озвученный в начале лекции, будет звучать так:

Для хранения объект с условием частого поиска, изменения, удаления, добавления элементов в этой коллекции по идентификатору этого объекта, в языке Java рекомендуется использовать коллекцию `HashMap`. Так как именно эта коллекция основана на хеш-таблицах в языке Java.

Визуально коллекцию `HashMap` можно представить как массив, в котором в качестве ключа используется идентификатор, а в качестве значения объект, связанный с этим идентификатором.

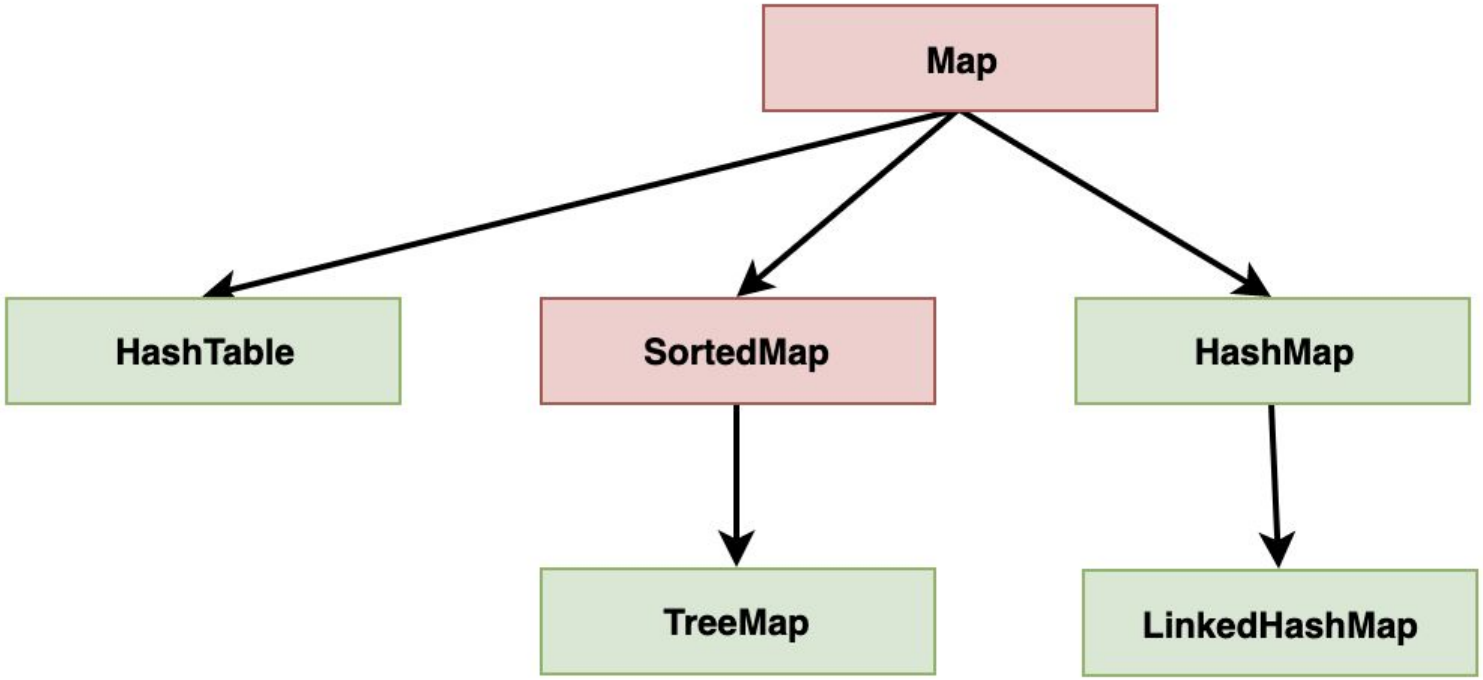
Коллекция HashMap



Номер ячейки массива, в котором будет храниться пара ключ-значение, вычисляется на основе хеш-кода ключа.

Интерфейс Map

Коллекция `HashMap` реализует базовый интерфейс `Map<K, V>`, который можно представить, как словарь или справочник, содержащий сопоставление ключей с соответствующими им значениями.



Ключевые методы интерфейса `Map`

- **put(k,v)** — добавить элемент `v` с ключом `k`;
- **putAll(otherMap)** — добавить все элементы из другой `Map`;
- **get(k)** — вернет значение по ключу `k`; `null` — если такого ключа нет;
- **remove(k)** — удаляет элемент с заданным ключом, напоследок возвращает его значение;
- **containsKey(k)** — вернет `true`, если такой ключ используется в `map`;
- **containsValue(v)** — вернет `true`, если значение хоть раз встречается в `map`. *Внимание, O(n)!*;

Ключевые методы интерфейса `Map`

- **size()** – количество элементов в map;
- **isEmpty()** – `size() == 0`;
- **clear()** – удалить всё;
- **keySet()** – возвращает `Set`, состоящий из ключей данной коллекции;
- **values()** – возвращает `Collection`, состоящий из значений данной коллекции;
- **entrySet()** – возвращает `Set`, состоящий из пар `<Ключ , Значение> (Entry)`.

Особенности реализации `HashMap`

Чтобы скорость выполнения операций вставки, удаления и поиска при работе с `HashMap` на произвольных классах языка Java была максимально приближена к O(1), необходимо переопределить методы `equals()` и `hashCode()`, соблюдая нижеописанный контракт.

При переопределении `equals` и `hashCode` в своих классах необходимо следить за соблюдением следующих свойств:

1. **вызов метода `hashCode` на одном и том же объекте должен возвращать одно и то же значение;**
2. **если два объекта одинаковые (`equals=true`), то у них должен быть одинаковый хеш-код;**
3. **если объекты не одинаковые, то не обязательно, чтобы хеш-код был разным. Но разный хеш для разных объектов значительно улучшает производительность при использовании хеш-таблиц.**

Вопрос

Что будет, если изменить ключ (объекта), добавленный в коллекцию `HashMap`? Например, изменить поле, которое участвует в вычислении `hashCode`?

```
class Driver {

    String name;

    String surname;

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (surname != null ? surname.hashCode() : 0);
        return result;
    }
}
```

Ответ

В этом случае в `HashMap` могут оказаться два одинаковых объекта, что может привести к трудно обнаруживаемым ошибкам.

Для решения такой проблемы в `HashMap` рекомендуется использовать **неизменяемые (immutable) объекты**: неизменяемым называется класс поля объектов, который нельзя изменить после создания.

Для создания неизменяемого объекта нужно, чтобы поля класса, из которого создается объект, были помечены модификатором `final`.

Пример переопределения метода `equals`

```
public boolean equals(Object obj) {
    // Сравним с собой
    if (obj == this) {
        return true;
    }
    // Проверим тип
    if (obj == null || obj.getClass() != this.getClass()) {
        return false;
    }
    // Приведем тип
    Type other = (Type) obj;
    // Сравним значащие поля (не забывая проверку на null)
    return id == other.id
        && (this.field1 == other.field1 || (this.field1 != null && this.field1.equals(other.getField1())))
        && ... ;
}
```

Пример переопределения метода hashCode

Вручную:

```
@Override
public int hashCode() {
    int result = 17; //Простое число
    result = 31 * result + field1.hashCode();
    result = 31 * result + intField;
    result = 31 * result + field2.hashCode();
    return result;
}
```

Используя стандартную библиотеку:

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2, intField);
}
```

Пример использования HashMap

Создадим коллекцию `HashMap`, где в качестве ключа используем тип строки (фамилию водителя), а в качестве значения тип `integer` (возраст водителя).

```
Map<String, Integer> ages = new HashMap<>();
ages.put("Иванов", 24);
ages.put("Петров", 27);
ages.put("Кузьмина", 24);
System.out.println(ages);
// {Иванов=24, Петров=27, Кузьмина=24}

ages.put("Иванов", 33);
System.out.println(ages);
// {Иванов=33, Петров=27, Кузьмина=24}

System.out.format("Возраст Петрова %d лет%n", ages.get("Петров"));
// Возраст Петрова 27 лет

System.out.println(ages.remove("Иванов"));
// 33
```

Обратите внимание, двух элементов с одинаковым ключом быть не может, значение будет перезаписано новым. А два одинаковых значения (по разным ключам) могут существовать.

Вопрос

Можно ли использовать `null` в качестве ключа?

Вопрос

Можно ли использовать `null` в качестве ключа?

Ответ

Да.

Пример добавления и извлечения объекта с ключем, равным `null`.

```
Map<String, Integer> ages = new HashMap<>();
ages.put("Иванов", 33);
ages.put("Петров", 27));

ages.put(null, 125);
// Добавили значение с ключем null

System.out.println(ages);
// {null=125, Иванов=33, Петров=27}
```

Отличие `HashMap` и `Hashtable`

Как и в случае с коллекциями `Stack` и `Vector`, `Hashtable` считается устаревшей и менее производительной, чем `HashMap`.

Еще одна задача

Дано:

Напишем метод для проверки наличия у водителя нужной категории вождения (A, B, C, D, E).

Вопрос:

Какую коллекцию стоит в этом случае использовать для хранения категорий, чтобы можно было быстро искать наличие нее у водителя?

Ответ

Как и в предыдущей задаче возможными вариантами решения могли бы быть массивы или списки. Но проблема медленного поиска (полный перебор) по-прежнему актуальна.

Можно попробовать ввести `HashMap`, но тогда у каждой категории должен быть обязательно добавлен идентификатор. Но что делать, если идентификатора нет или объект небольшой? Если мы будем использовать в качестве ключа все поля объекта в коллекции, по факту мы будем использовать сам объект как ключ.

Например, если мы храним строки в коллекции, то и ключ, и значение будут строки. Избыточно хранить две строки для одного небольшого объекта. Если объект небольшой, то проще хранить только одну строку. В этом случае используют **множества**. Это еще один вид коллекции для хранения.

Реализация коллекции `HashSet`

Множество

В математике: набор или совокупность объектов, которые обладают общим для них свойством. Над математическим множеством можно выполнять операции: объединение, пересечение, разность и т.д.

В программировании: структура данных, которая хранит объекты определенного типа. При этом порядок объектов не поддерживается. Обычно позволяет выполнять стандартные математические операции над множествами.

В Java структура данных множества определяет коллекцию (массив) неповторяющихся элементов: **{ a, b, c, d, e }** — множество символов.

Одной из наиболее часто используемой коллекцией, описывающей множества, является коллекция `HashSet`, которая в свою очередь основана на `HashMap` и как следствие основана на хеш-таблицах.

Внутренняя реализация `HashSet`

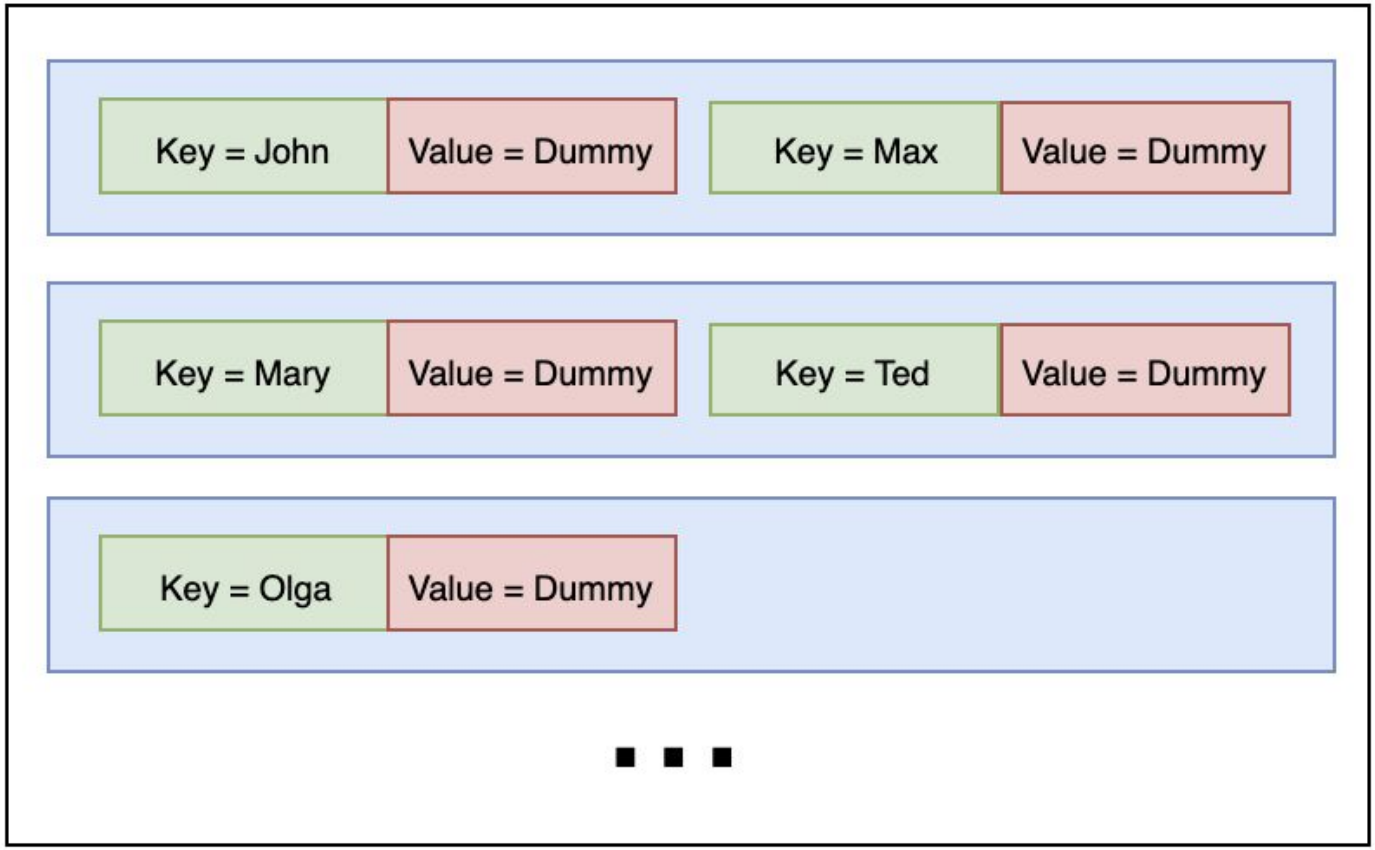
`HashSet` делегирует все свои методы объекту `HashMap`, созданному внутри своей реализации. В качестве ключа для этой коллекции используется сам объект, а в качестве значения — объект заглушка.

Код из JDK

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;

    private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    /**
     * Constructs a new, empty set; the backing {@code HashMap} instance has
     * default initial capacity (16) and load factor (0.75).
     */
    public HashSet() { map = new HashMap<>(); }
```

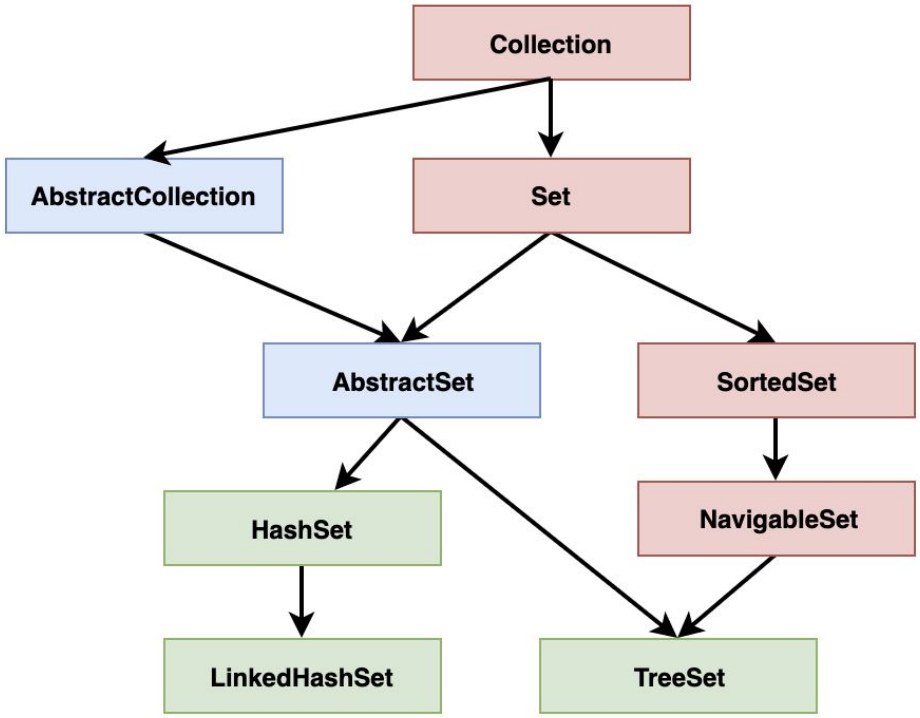
Представление HashSet

(Dummy — объект заглушка)



Интерфейс `Set`

`HashSet` – это класс, коллекция неповторяющихся элементов, реализующая интерфейс `Set`.



Ключевые методы интерфейса `Set`

- **add(e)** — добавляет элемент в множество, если такого еще нет. Возвращает `true` при успешном добавлении.
- **addAll(collection)** — добавляет все элементы из коллекции в множество, если такого еще нет. Возвращает `true` при успешном добавлении.
- **clear()** — удаляет все элементы из множества.
- **contains(o)** — возвращает `true`, если переданный объект есть в множестве.
- **containsAll(collection)** — возвращает `true`, если все объекты из переданной коллекции есть в множестве.
- **isEmpty()** — возвращает `true`, если сет пустой.

Ключевые методы интерфейса `Set`

- **iterator()** — возвращает итератор по элементам сета.
- **remove(e)** — удаляет элемент из множества. Вернет `true`, если такой элемент был.
- **removeAll(collection)** — удаляет из множества все элементы, которые есть в коллекции. Вернет `true`, если сет был изменен в результате вызова метода.
- **retainAll(collection)** — оставляет в сете только элементы, которые есть в коллекции, удаляет все остальные. Вернет `true`, если сет был изменен в результате вызова метода.
- **size()** — возвращает количество элементов в множестве.
- **toArray()** — возвращает массив, содержащий все элементы множества.

Пример использования HashSet

Пример использования класса `String` в качестве типа объектов хранения:

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("B");
System.out.println(set.contains("A"));
// true
System.out.println(set);
// [A, B]
set.remove("A");
System.out.println(set);
// [B]
```

Сравнение `HashSet` с `ArrayList`

	HashSet	ArrayList
Сохраняет порядок элементов	Нет	Да
Можно хранить дубликаты	Нет	Да
Скорость поиска	O(1)	O(n)

Дополнительные материалы

Java собеседование. Коллекции с пункта 7 по 14

Что было изучено

- Интерфейс `Map` и его реализации `HashMap`;
- Интерфейс `Set` и его реализации `HashSet`;
- Хеш-таблицы;
- Переопределение `hashCode` и `equals`;
- Иммутабельность и коллизии.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Юрий Пеньков

Юрий Пеньков