Коллекции в Java

Антон Авдеев

SIMBIRSOFT
software development

Что такое коллекции?

Классы позволяющие хранить и производить операции над множеством объектов.

java.lang.Iterable

- java.util.Collection
  - java.util.List - список
  - java.util.Set - множество
  - java.util.Queue - очередь
- java.util.Map - карта, ассоциативный массив

Зачем нужны коллекции?

Array vs Collection

- Типизация данных

- Безопасность типа хранимых данных на уровне компилятора

- Фиксированный размер массива

- Экономия памяти (примитивы)

- Быстрый доступ к каждому элементу

- f.length

- Generics, либо Objects

- Generics, либо приведение типов

- Размер не фиксирован

- Работа только с объектами

- Скорость доступа зависит от способа имплементации коллекции

- f.size()

Методы коллекций

Определение размера

int size(); — количество элементов

boolean isEmpty(); — проверка на пустоту

```
Collection col = new ArrayList();

If (col.isEmpty()) { …    как правильно проверять пустоту
        коллекции?
If (col.size() != 0) {…
```

Проверки на вхождение

boolean contains(Object o); — одного элемента

boolean containsAll(Collection c); — всех элементов коллекции c

Методы коллекций

Добавление элементов

boolean add(E e); — одного элемента

boolean addAll(Collection c); — элементов коллекции

Удаление элементов

boolean remove(Object o); — одного элемента

boolean removeAll(Collection c); — элементов коллекции

boolean retainAll(Collection c); — удаление элементов не из коллекции c

void clear(); — удаление всех элементов

Преобразование в массив

Object[] toArray(); — создает новый массив

Object[] toArray(Object[] a); — использует переданный массив

```
list.add(“1”);
Foo[] foos = (Foo[]) list.toArray(new Foo[0]);
list.add(“2”);
```

```
Collection list = java.util.Arrays.asList(foos);
```

Итерирование

java.lang.Iterable:

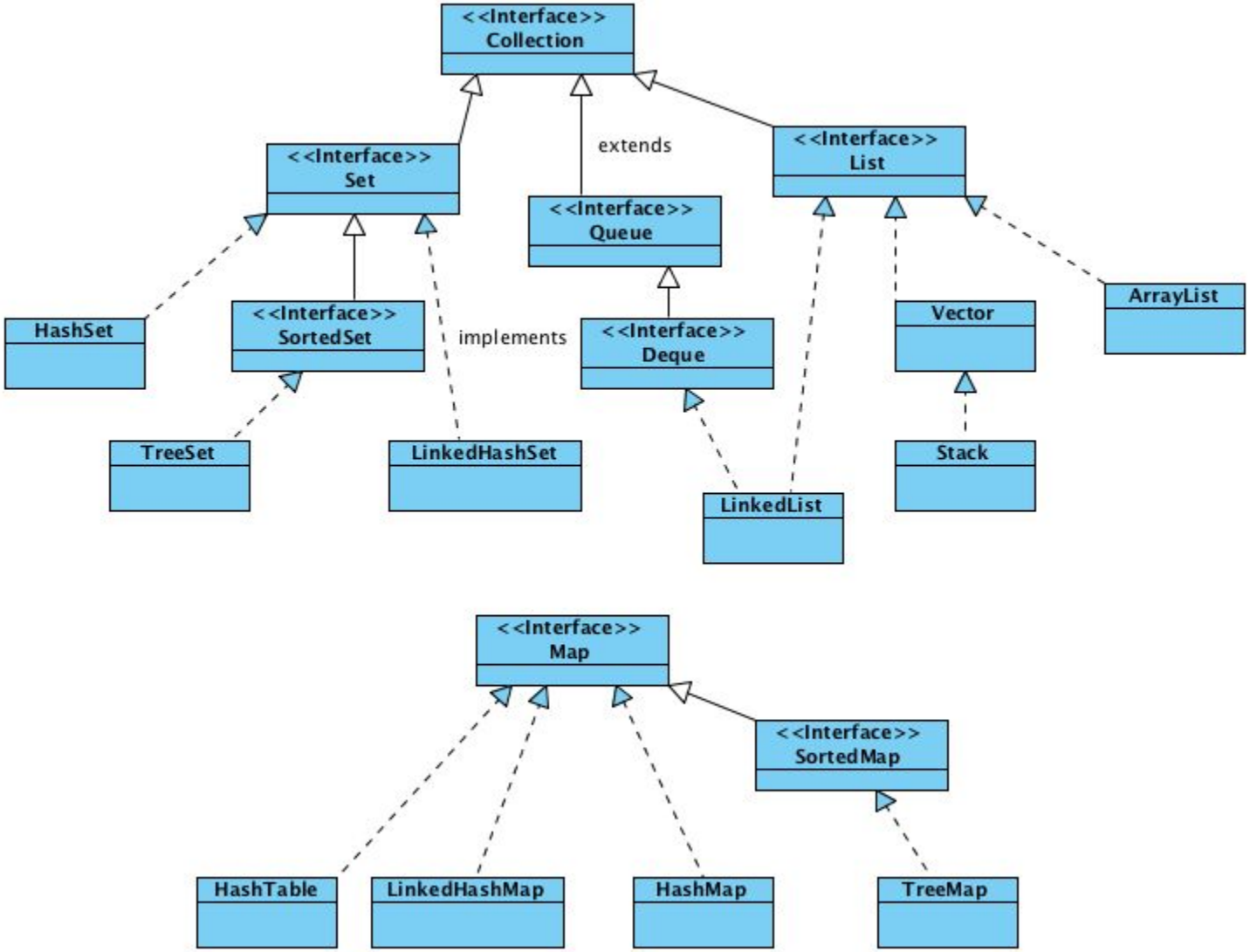
методы Iterator<T> iterator();

- **boolean** hasNext();
- T next();
- **void** remove();

```
List list = new ArrayList(30);

for (Object o : list) {
    System.out.println(o.toString());
}

for (Iterator iter = list.iterator(); iter.hasNext();) {
    System.out.println(iter.next().toString());
}
```

Сколько итераций будет выполнено?

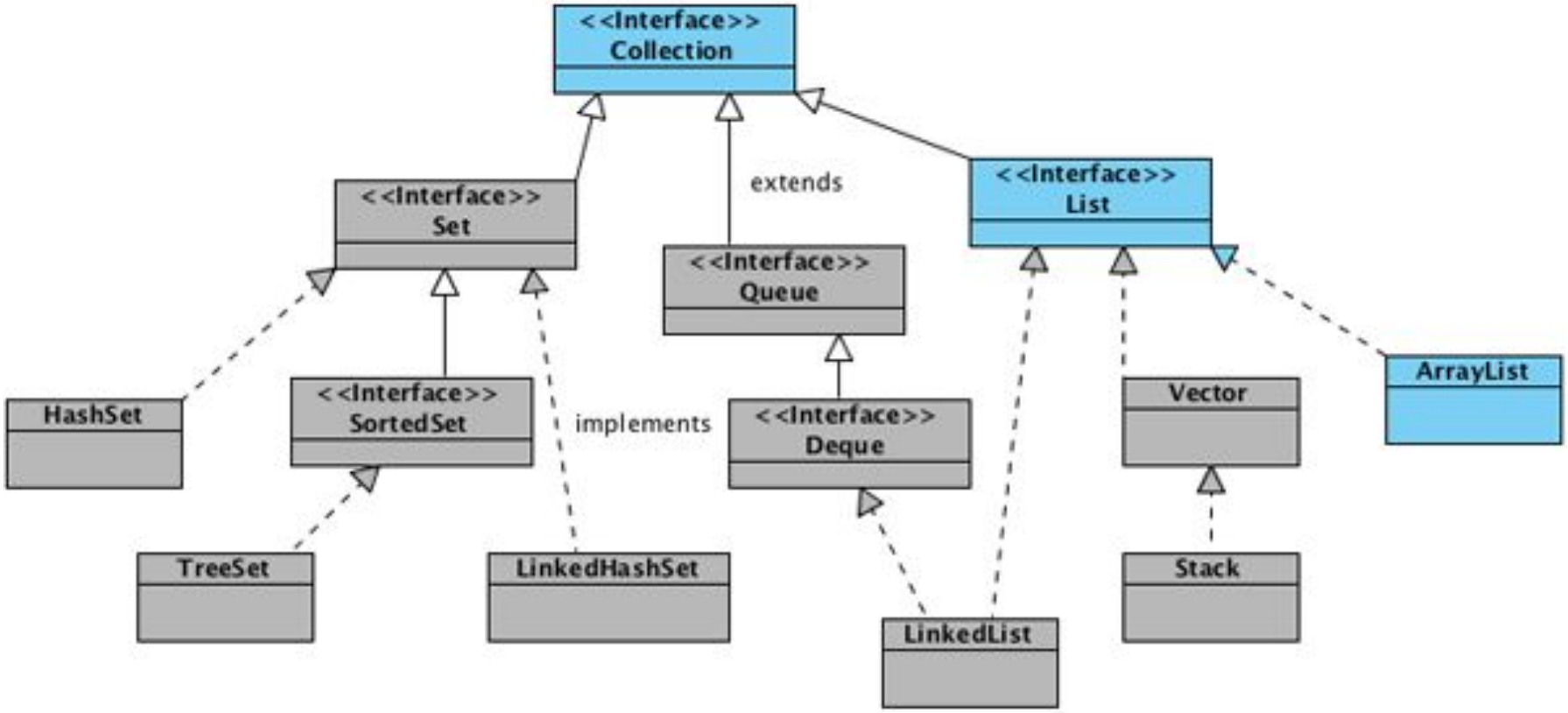
Стандартные коллекции



Стандартные коллекции

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order

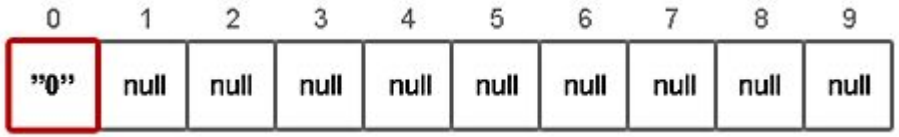
ArrayList


<<Interface>>
Collection
<<Interface>>
Set
extends
<<Interface>>
List
<<Interface>>
Queue
HashSet
<<Interface>>
SortedSet
implements
<<Interface>>
Deque
Vector
ArrayList
TreeSet
LinkedHashSet
LinkedList
Stack

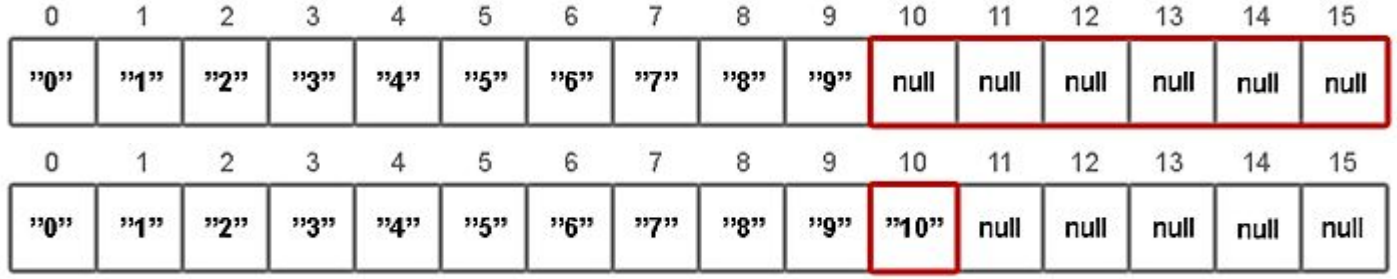
ArrayList

ArrayList может менять свой размер во время исполнения программы
Элементы ArrayList могут быть абсолютно любых типов в том числе и null

```
ArrayList<String> list = new ArrayList<>();
list.add("0");
```



При добавлении 11-го элемента



ArrayList

Добавление в «середину» списка происходит в три этапа:

1) проверяется, достаточно ли места в массиве;
2) подготавливается место для нового элемента с помощью **System.arraycopy()**;
3) перезаписывается значение у элемента с указанным индексом.

Удалять элементы можно двумя способами:
— по индексу **remove(index)**
— по значению **remove(value)**

ArrayList

Итоги

— Быстрый доступ к элементам по индексу за время O(1);

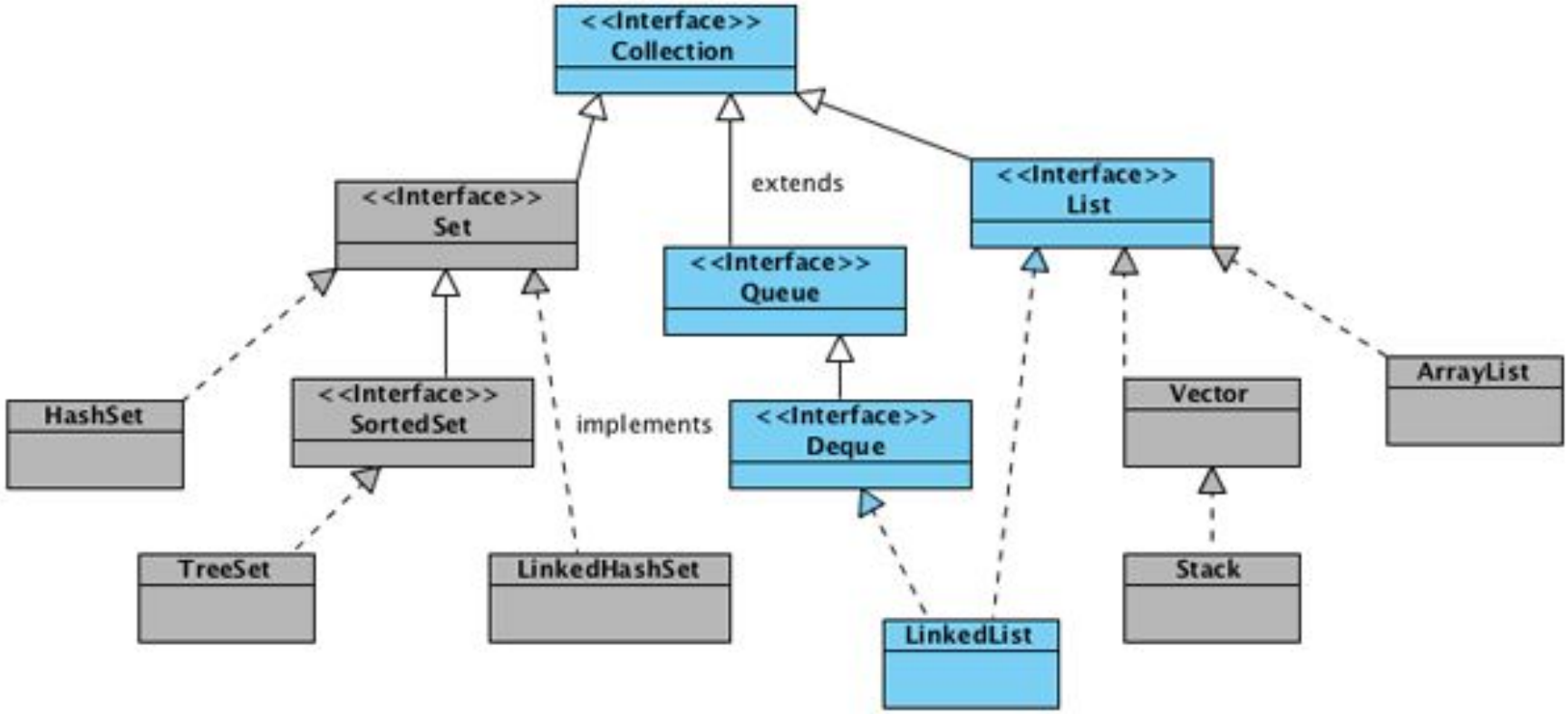
— Доступ к элементам по значению за линейное время O(n);

— Медленный, когда вставляются и удаляются элементы из «середины» списка;

— Позволяет хранить любые значения в том числе и null;

— Не синхронизирован.

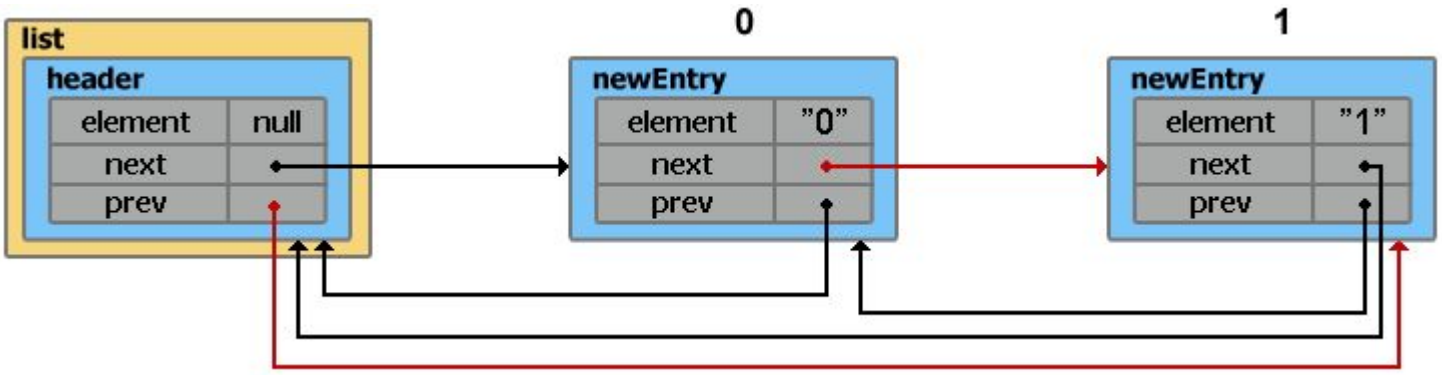
LinkedList


<<Interface>>
Collection
<<Interface>>
Set
extends
<<Interface>>
List
<<Interface>>
Queue
HashSet
<<Interface>>
SortedSet
implements
<<Interface>>
Deque
Vector
ArrayList
TreeSet
LinkedHashSet
LinkedList
Stack

LinkedList

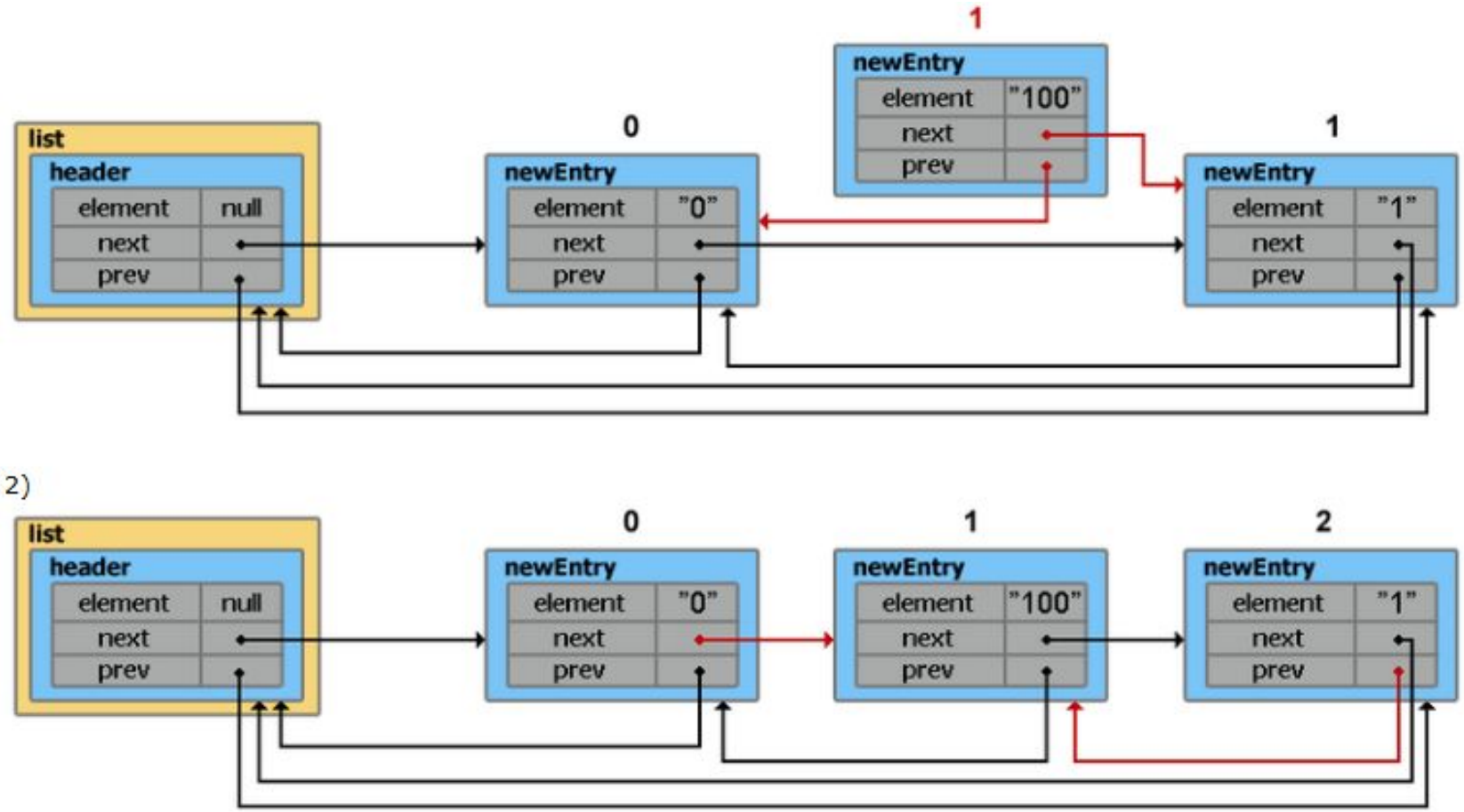
Является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы.
Итератор поддерживает обход в обе стороны.
Реализует методы получения, удаления и вставки в начало, середину и конец списка.
Позволяет добавлять любые элементы в том числе и null.

```
List<String> list = new LinkedList<>();
list.add("0");
list.add("1");
```



LinkedList

Добавление элементов в «середину» списка



LinkedList. Применение (FIFO)

```
public class Queue {

    private final LinkedList list = new LinkedList();

    public void put(Object v) {
        list.addFirst(v);
    }

    public Object get() {
        return list.removeLast();
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }
}
```

LinkedList

Итоги

— Из LinkedList можно организовать стэк, очередь, или двойную очередь, со временем доступа O(1);

— На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n). Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1);

— Позволяет добавлять любые значения в том числе и null. Для хранения примитивных типов использует соответствующие классы-оберки;

— Не синхронизирован.

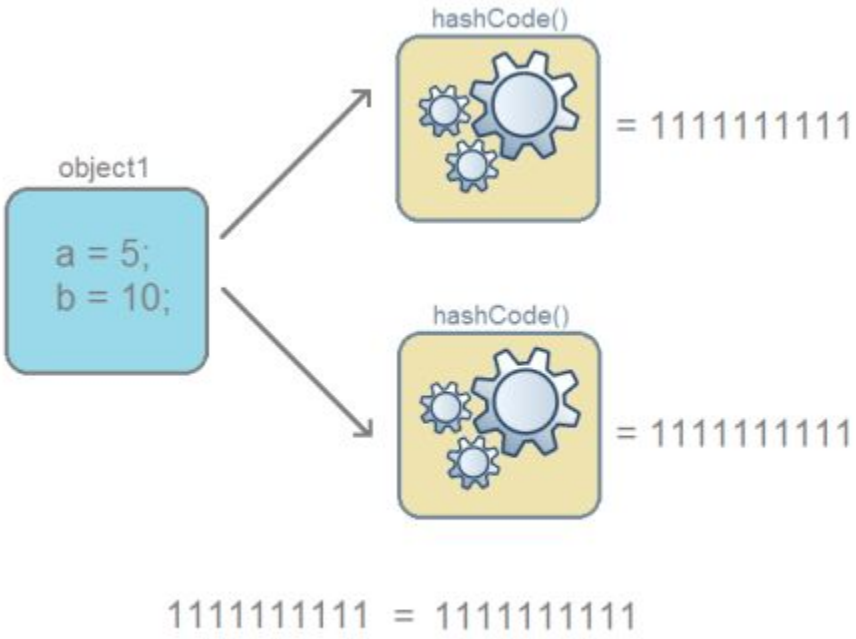
HashCode

Если очень просто, то хеш-код — это число. Если более точно, то это битовая строка фиксированной длины, полученная из массива произвольной длины.

Ситуация, когда у разных объектов одинаковые хеш-коды называется — **коллизией**. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

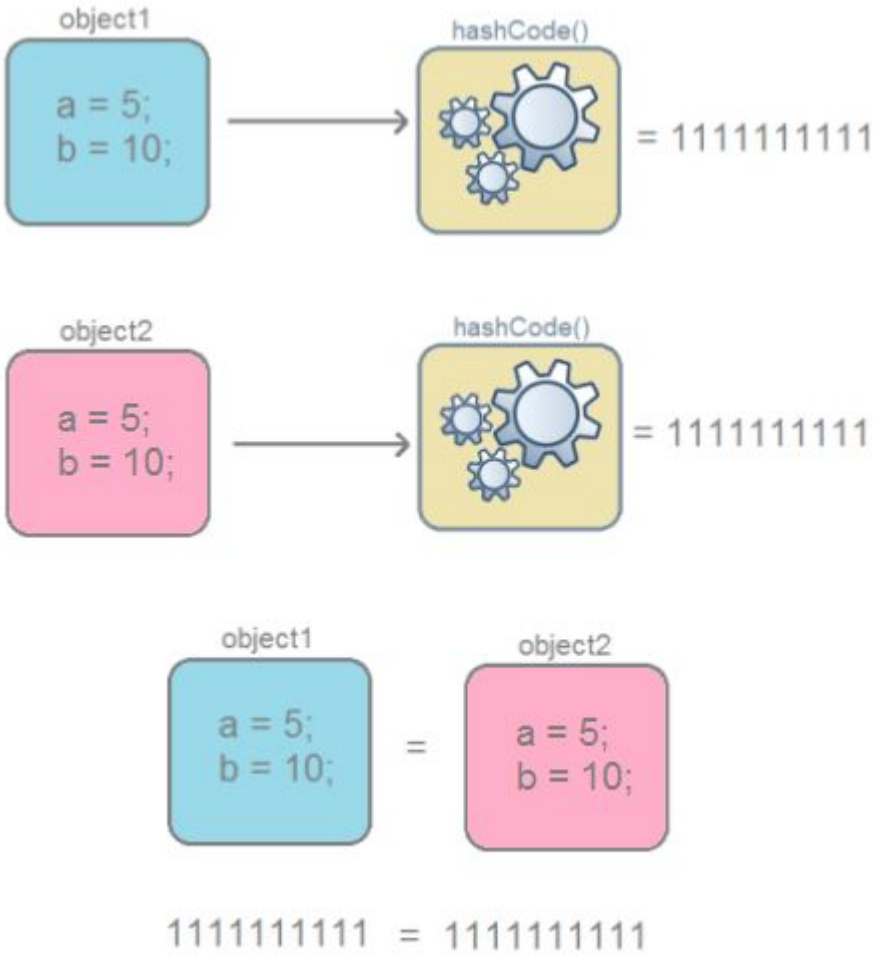
HashCode

1. Для одного и того-же объекта, хеш-код всегда будет одинаковым



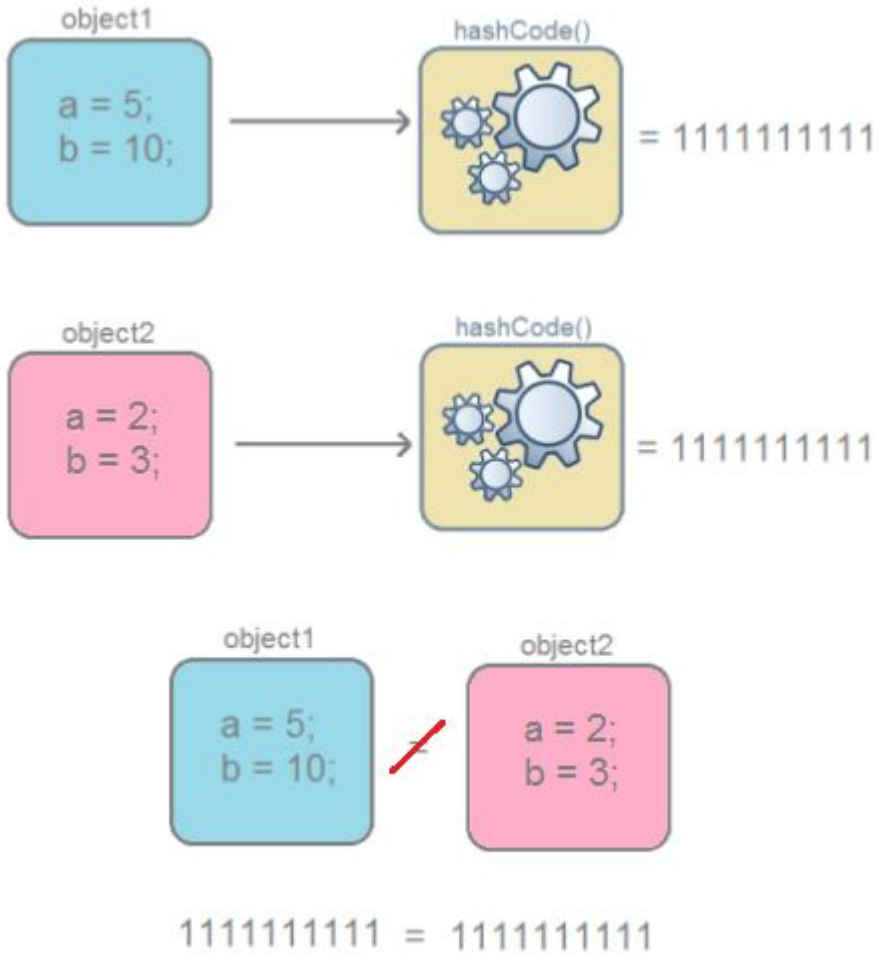
HashCode

2. Если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот, см. правило 3).



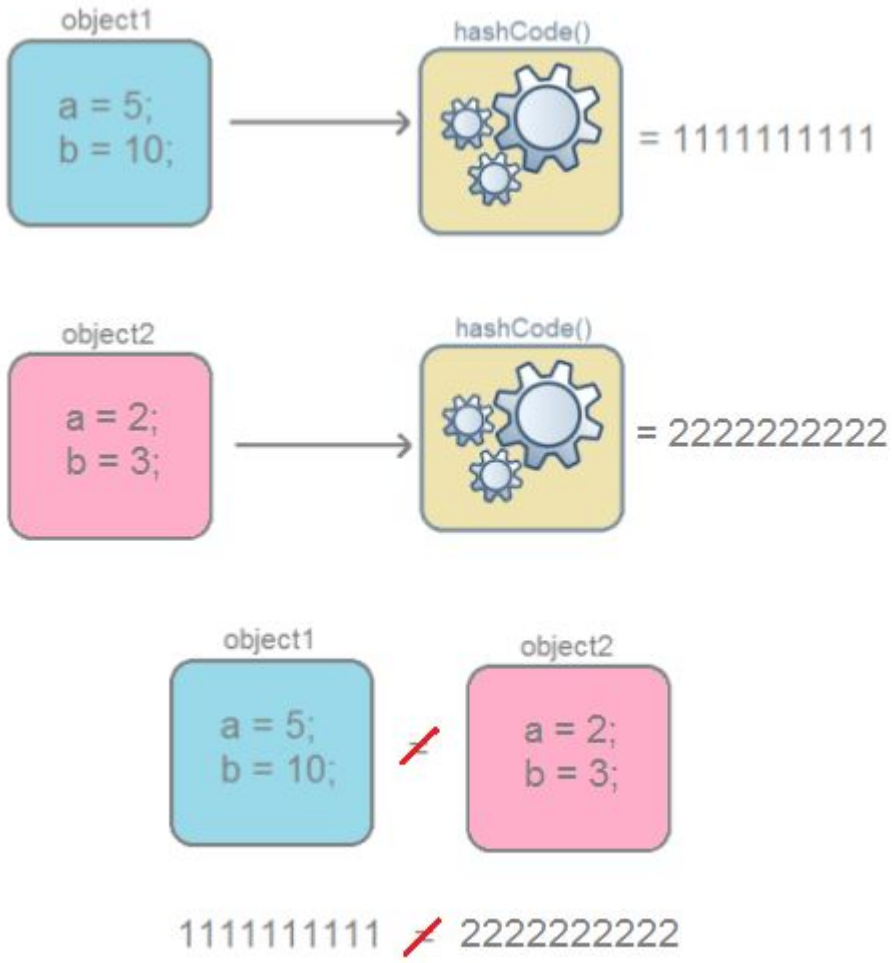
HashCode

3. Если хеш-коды равны, то входные объекты не всегда равны (коллизия).



HashCode

4. Если хеш-коды разные, то и объекты гарантированно разные.



Эквивалентность

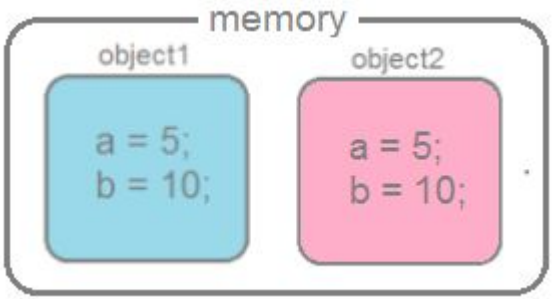
Каждый вызов оператора **new** порождает новый объект в памяти.

```
public class BlackBox {

    private int varA;
    private int varB;

    public BlackBox(int varA, int varB) {
        this.varA = varA;
        this.varB = varB;
    }
}
```

Эквивалентность

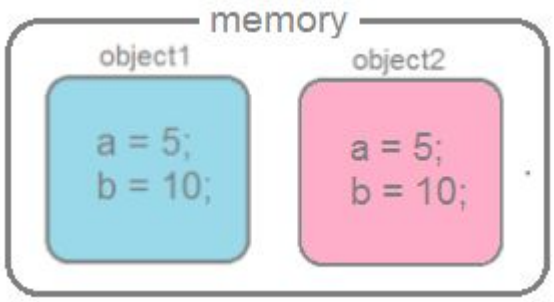
Создадим класс для демонстрации *BlackBox*.

```
public class DemoBlackBox {

    public static void main(String[] args) {
        BlackBox object1 = new BlackBox(5, 10);
        BlackBox object2 = new BlackBox(5, 10);
    }
}
```



Эквивалентность

Содержимое этих объектов одинаково, то есть эквивалентно. Для проверки эквивалентности в классе **Object** существует метод **equals()**, который сравнивает содержимое объектов и выводит значение типа **boolean true**, если содержимое эквивалентно, и **false** — если нет.




```
object1.equals(object2);// должно быть true,
поскольку содержимое объектов эквивалентно
object1.hashCode() == object2.hashCode()//
должно быть true
```

Что выведется на экран в первом случае? Во втором?

Эквивалентность

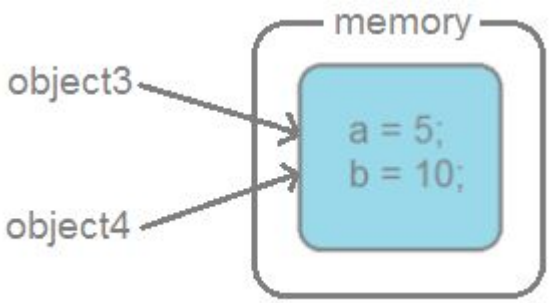
```
public boolean equals(Object obj) {
    return (this == obj);
}
```

При сравнение объектов, операция “==” вернет **true** лишь в одном случае — когда ссылки указывают на один и тот-же объект. В данном случае не учитывается содержимое полей.

```
public native int hashCode();
```

Эквивалентность

```
public class DemoBlackBox {

    public static void main(String[] args) {
        ...
        BlackBox object3 = new BlackBox(5, 10);
        BlackBox object4 = object3; // Переменная object4 ссылается на тот-же объект что и переменная object3
        object3.equals(object4); // true
    }
}
```



Эквивалентность

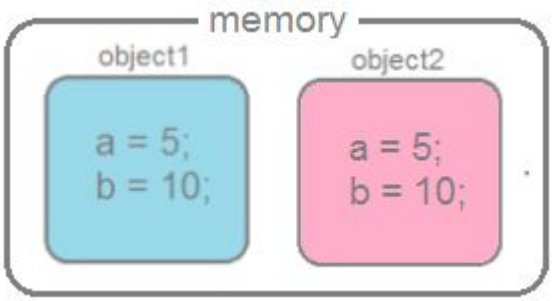
```
public class BlackBox {

    ...

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + varA;
        result = prime * result + varB;
        return result;
    }
}
```

Эквивалентность

```
public class BlackBox {
    ...
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        BlackBox other = (BlackBox) obj;
        if (varA != other.varA)
            return false;
        if (varB != other.varB)
            return false;
        return true;
    }
}
```

Эквивалентность

```
object1.equals(object2);
object1.hashCode() == object2.hashCode();
```



Что выведется на экран в первом случае? Во втором?

Эквивалентность

Требования к реализации equals()

reflexive. x.equals(x) == true

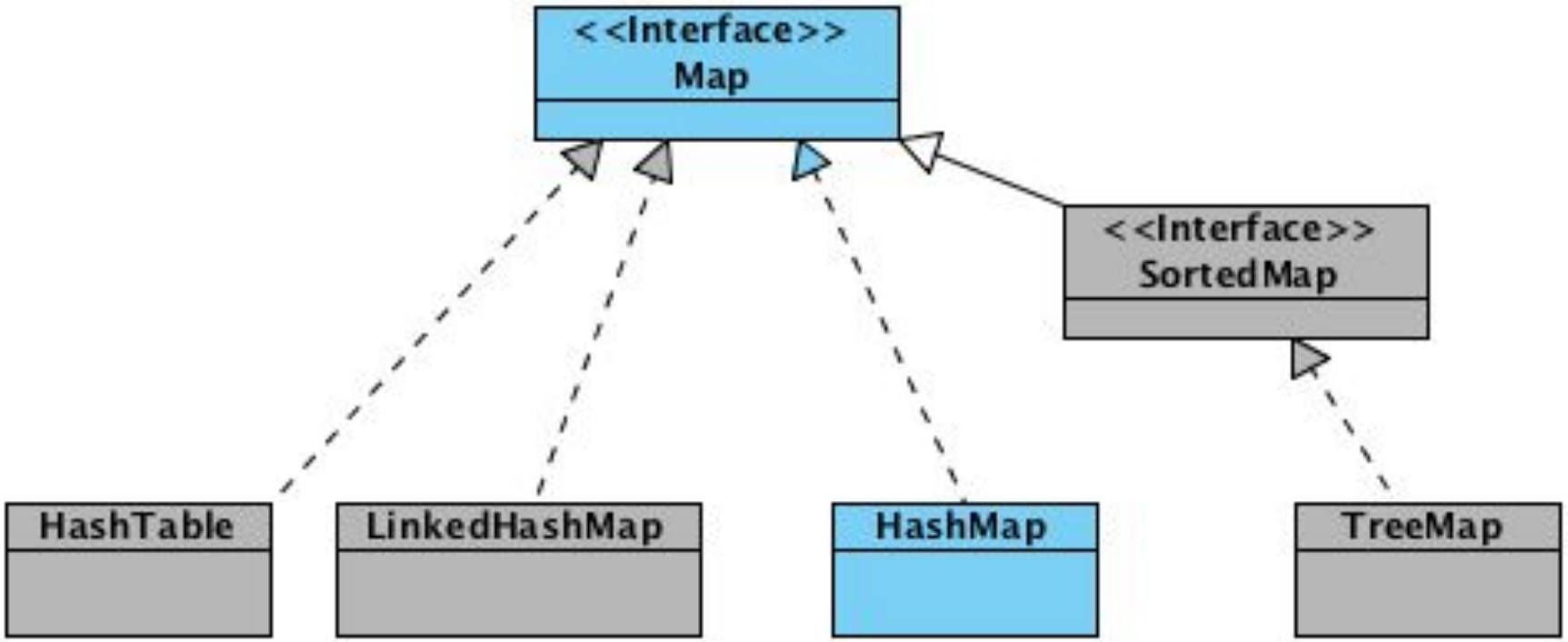
symmetric. Если x.equals(y) == true, то y.equals(x) должен быть тоже true.

transitive. Если x.equals(y) == true и y.equals(z) == true, то результат x.equals(z) тоже должен быть true

consistent. Для любых объектов x и y, если их содержимое не изменяется, то множественный вызов x.equals(y) должен возвращать одно и тоже значение

Для x.equals(null) == false.

HashMap


<<Interface>>
Map
<<Interface>>
SortedMap
HashTable
LinkedHashMap
HashMap
TreeMap

HashMap

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

```
Map<String, String> hashmap = new HashMap<>();
```

Каждый HashMap содержит ряд свойств:

table — массив типа **Entry[]**, который является хранилищем ссылок на списки (цепочки) значений;

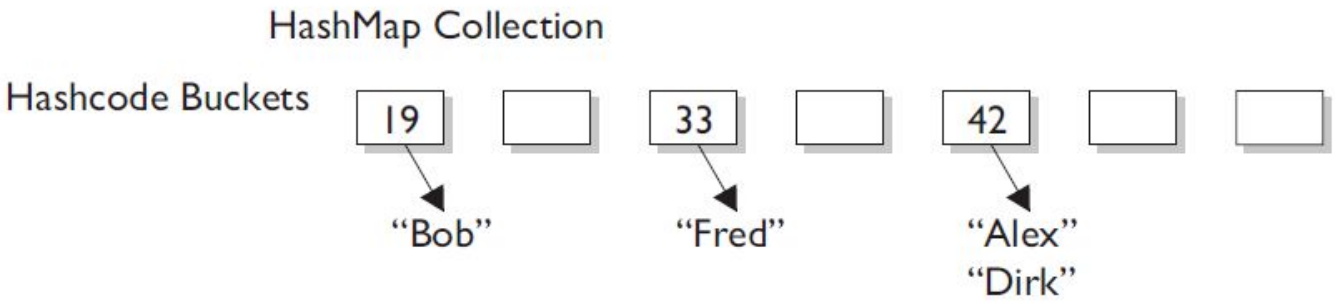
threshold — предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое.

size — количество элементов HashMap-а;

HashMap

hashCode()

Key	Hashcode Algorithm	Hashcode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) + I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + D(4)	= 33



Поиск элемента по ключу происходит в 2 этапа:

1) Поиск нужного контейнера (используя hashCode())

2) Поиск элемента в контейнере (используя equals())

HashMap

Добавление, получение, удаление элементов

```
hashmap.put("0", "zero");
hashmap.get(key);
hashmap.remove(key);
```

Итераторы

```
// 1
for (Map.Entry<String, String> entry : hashmap.entrySet())
    System.out.println(entry.getKey() + " = " + entry.getValue());

// 2
for (String key : hashmap.keySet())
    System.out.println(hashmap.get(key));

// 3
Iterator<Map.Entry<String, String>> itr = hashmap.entrySet().iterator();
while (itr.hasNext())
    System.out.println(itr.next());
```

HashMap

Итоги

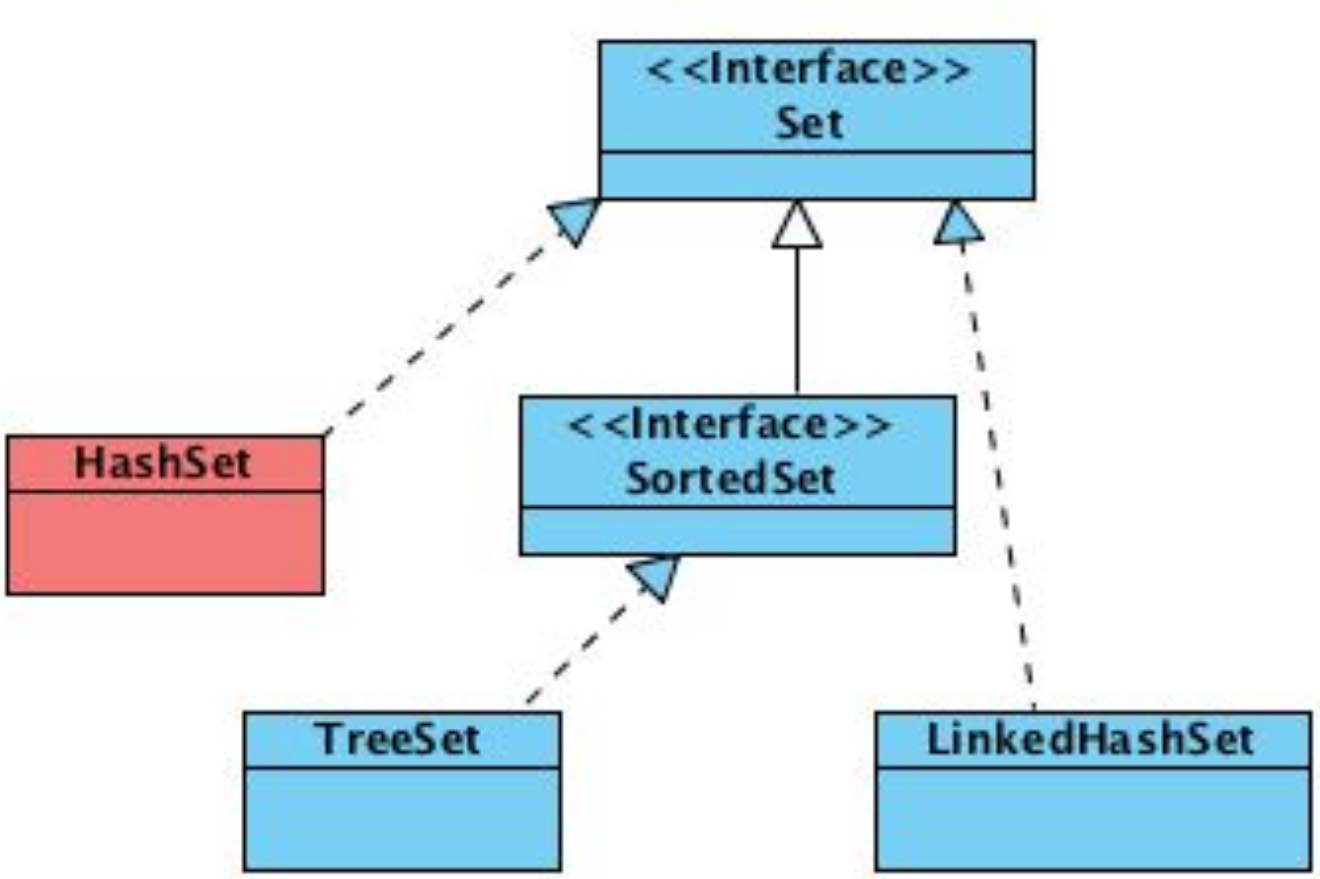
— Добавление элемента выполняется за время O(1);

— Операции получения и удаления элемента могут выполняться за время O(1), если хэш-функция равномерно распределяет элементы и отсутствуют коллизии;

— Ключи и значения могут быть любых типов, в том числе и null. Для хранения примитивных типов используются соответствующие классы-оберки;

— Не синхронизирован.

HashSet


<<Interface>>
Set
HashSet
<<Interface>>
SortedSet
TreeSet
LinkedHashSet

HashSet

В множествах **Set** каждый элемент хранится только в одном экземпляре, а разные реализации Set используют разный порядок хранения элементов.

Методы аналогичны методам **ArrayList** за исключением того, что метод **add(Object o)** добавляет объект в множество только в том случае, если его там нет.

```
Set<Integer> intSet = new HashSet<>();
intSet.add(1);
intSet.add(4);
intSet.add(3);
intSet.add(1);
intSet.add(2);
intSet.add(3);
intSet.add(2);
```

[1, 2, 3, 4]

[3, 1, 4, 2]

[2, 4, 1, 3]

Лабораторная работа

Задание: Вычислить сколько раз каждая буква встречается в тексте.

```
public class UniqueChars {

    private Map<Character, Integer> map = new HashMap<>();
    private String text;

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    ...
}
```

Лабораторная работа

```
public class UniqueChars {
    ...

    public void calculate() {
        for (char c : text.toCharArray()) {
            if (Character.isLetter(c)) {
                if (map.containsKey(c)) {
                    map.put(c, map.get(c) + 1);
                } else {
                    map.put(c, 1);
                }
            }
        }
    }

    ...
}
```

Лабораторная работа

```
public class UniqueChars {
    ...

    @Override
    public String toString() {
        String result = "";
        for (Entry<Character, Integer> entry : map.entrySet()) {
            result += "char: " + entry.getKey() +
                "; count: " + entry.getValue() + "\n";
        }
        return result;
    }
}
```

Лабораторная работа

- Отдельный подсчет цифр;
- Подсчет в указанном регистре (верхнем, нижнем, не учитывать);

- Реализовать класс корзины интернет магазина по следующему интерфейсу:

```
interface Basket {

    void addProduct(String product, int quantity);

    void removeProduct(String product);

    void updateProductQuantity(String product, int quantity);

    void clear();

    List<String> getProducts();

    int getProductQuantity(String product);

}
```

Лабораторная работа

```
interface Smartable {

    public List<Integer> removeInRange(List<Integer> list, int element, int start, int end);

    public boolean isUnique(Map<String, String> map);

    public Map<String, Integer> intersect(Map<String, Integer> map1, Map<String, Integer> map2);

    public int countCommon(List<Integer> list1, List<Integer> list2);

    public Set<String> removeEvenLength(Set<String> set);

    public int maxOccurrences(List<Integer> list);
}
```

Лабораторная работа

public List<Integer> removeInRange(List<Integer> list, **int** element, **int** start, **int** end)

Который принимает на вход List<Integer>, значение, стартовый и конечный индекс).

Метод должен удалить все вхождения **element** между **start** (включительно) и **end** (исключительно) индексами. Вхождения вне этого индекса, а также другие значения должны остаться без изменений.

Например, если для списка

(0, 0, 2, 0, 4, **0**, 6, **0**, 8, **0**, 10, **0**, 12, 0, 14, 0, 16) вызвать метод removeInRange(list, 0, 5, 13) результат будет

(0, 0, 2, 0, 4, 6, 8, 10, 12, 0, 14, 0, 16).

Лабораторная работа

```
public boolean isUnique(Map<String, String> map);
```

Который возвращает **true**, если в отображении нет двух и более одинаковых *value*, и **false** в противном случае.

Для пустого отображения метод возвращает **true**.

Например, для метода {Вася=Иванов, Петр=Петров, Виктор=Сидоров, Сергей=Савельев, Вадим=Викторов} метод вернет **true**,

а для {Вася=**Иванов**, Петр=**Петров**, Виктор=**Иванов**, Сергей=Савельев, Вадим=**Петров**} метод вернет false.

Лабораторная работа

```
public Map<String,Integer> intersect(Map<String, Integer> map1,
Map<String, Integer> map2);
```

Который возвращает отображение, который содержит пары «ключ=значение», присутствующие в обоих отображениях.

Например, для отображений {Janet=87, **Logan=62**, Whitaker=46, Alyssa=100, **Stefanie=80**, **Jeff=88**, **Kim=52**, Sylvia=95}

и {**Logan=62**, **Kim=52**, Whitaker=52, **Jeff=88**, **Stefanie=80**, Brian=60, Lisa=83, Sylvia=87}

Метод вернет {Logan=62, Stefanie=80, Jeff=88, Kim=52} (не обязательно в этом порядке)

Лабораторная работа

```
public int countCommon(List<Integer> list1, List<Integer> list2);
```

Который возвращает количество **уникальных** совпавших значений в обоих списках.

Например, для списков [3, 7, 3, -1, 2, 3, 7, 2, 15, 15] и [-5, 15, 2, -1, 7, 15, 36] метод вернет 4, т.к. совпали элементы -1, 2, 7 и 15.

*Обратите внимание, что второй раз **15** не считаются, т.к. нужно совпадение уникальных значений.*

Лабораторная работа

```
public Set<String> removeEvenLength(Set<String> set);
```

Который возвращает множество, в котором удалены все элементы четной длины из исходного множества.

Например, для множества ["foo", "buzz", "bar", "fork", "bort", "spoon", "!", "dude"] метод вернет ["foo", "bar", "spoon", "!"]

Лабораторная работа

```
public int maxOccurrences(List<Integer> list);
```

Который возвращает количество вхождений наиболее часто встречающегося элемента.

Например, для массива [**4**, 7, **4**, -1, 2, **4**, 7, 2, 15, 15] метод вернет 3, т.к. наиболее часто встречающееся значение **4** имеет 3 вхождения в массив.

Источники

https://google.ru
http://docs.oracle.com/javase/8/docs/api/index.html
https://habrahabr.ru/hub/java