Коллекции. Часть I

Croc Java School

hashCode & equals

Сравнение по значению

Рассмотрим класс, задающий координаты точки.

```
class Point {

  private int x;
  private int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Сравнение по значению

Как понять, что у двух точек одинаковые координаты.

```
Point p1 = new Point(5, 5);
Point p2 = new Point(5, 5);

System.out.println(p1 == p2); // false
```

Оператор == в данном случае определяет равенство двух ссылок.

Сравнение по значению

Ок, мы слышали про метод `equals`.

```
Point p1 = new Point(5, 5);
Point p2 = new Point(5, 5);

System.out.println(p1.equals(p2)); // false
```

По умолчанию используется реализация `equals` из родительского класса `Object`, которая не знает про внутреннее устройство класса `Point`.

Object.equals

Стандартная реализация `equals` в классе `Object`:

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

По умолчанию `equals` сравнивает ссылки и это не то поведение, которое нам нужно.

Переопределение equals

```
class Point {

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (!(obj instanceof Point))
      return false;
    Point point = (Point)obj;
    return x == point.x && y == point.y;
  }
}
```

Переопределение equals

1

```
if (this == obj)
  return true;
```

Если объекты равны по ссылке, то они равны и по значению.

Переопределение equals

2

```
if (!(obj instanceof Point))
  return false;
```

Если классы объектов разные, то они не равны по значению.

Переопределение equals

3

```
Point point = (Point)obj;
return x == point.x && y == point.y;
```

Если все значимые поля объектов совпадают, то эти объекты равны по значению.

Сравнение по значению

Вернемся к примеру.

```
Point p1 = new Point(5, 5);
Point p2 = new Point(5, 5);

System.out.println(p1.equals(p2)); // true
```

Теперь вызывается переопределенный метод equals и точки сравниваются по координатам.

Хеширование

Преобразование набора данных произвольной длины в битовую строку фиксированной длины.

Алгоритм преобразования называется хеш-функцией или функцией свертки.

$h(K) = m \in M$

K — исходный набор данных (например, массив чисел)
M — множество хешей фиксированного размера

Хеш-функция

На практике “битовая строка фиксированной длины” — это либо обычная строка определенного размера, либо число.

Пример простейшей хеш-функции:

```
h(int n) = n mod 17
```

Хеш-функция

Хеши двух одинаковых значений всегда совпадают.
Но они могут совпадать и для двух разных значений. В этом случае происходит коллизия хешей.

```
h(int n) = n mod 17
h(2) = 2 mod 17 = 2
h(19) = 19 mod 17 = 2
```

“Хорошая” хеш-функция

Критерии качества хеш-функции:

- быстрота вычисления
- минимизация коллизий

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to. [...] A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.
Thomas H. Cormen. Introduction to Algorithms, 3 Ed.

Object.hashCode

Класс `Object` определяет метод, позволяющий вычислить хеш объекта в виде значения типа `int`.

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

Что по смыслу возвращает реализация по умолчанию, не определено.
(Задается параметром JVM: `-XX:hashCode`).

Зачем нам это знать?

Равные объекты должны иметь равные хеши

Комментарий к `Object.equals`:

```
Note that it is generally necessary to override the {@code hashCode}
method whenever this method is overridden, so as to maintain the
general contract for the {@code hashCode} method, which states
that equal objects must have equal hash codes.
```

Другими словами:

если `x.equals(y)`, то `x.hashCode() == y.hashCode()`

Выполняется ли это условие для класса Point?

```
Point p1 = new Point(5, 5);
Point p2 = new Point(5, 5);

System.out.println(p1.equals(p2)); // true
System.out.println(p1.hashCode() == p2.hashCode()); // false
```

Это нарушение не позволит использовать наш класс в коллекциях, основанных на хешировании: `HashMap`, `HashSet` и проч. (см. дальше).

Переопределение hashCode

```
class Point {

  @Override
  public int hashCode() {
    return Objects.hash(x, y);
  }
}
```

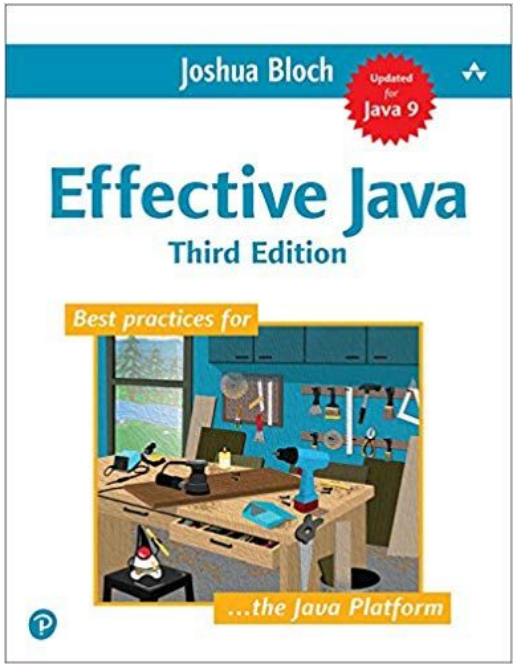
Любознательные могут изучить реализацию метода `Objects.hash` самостоятельно.

Теперь все в порядке

```
Point p1 = new Point(5, 5);
Point p2 = new Point(5, 5);

System.out.println(p1.equals(p2)); // true
System.out.println(p1.hashCode() == p2.hashCode()); // true
```

Класс `Point` готов к использованию в коллекциях.

Почитать на тему equals & hashCode



Joshua Bloch. Effective Java, 3 Ed.
Item 9: Always override hashCode when you override equals

Коллекции

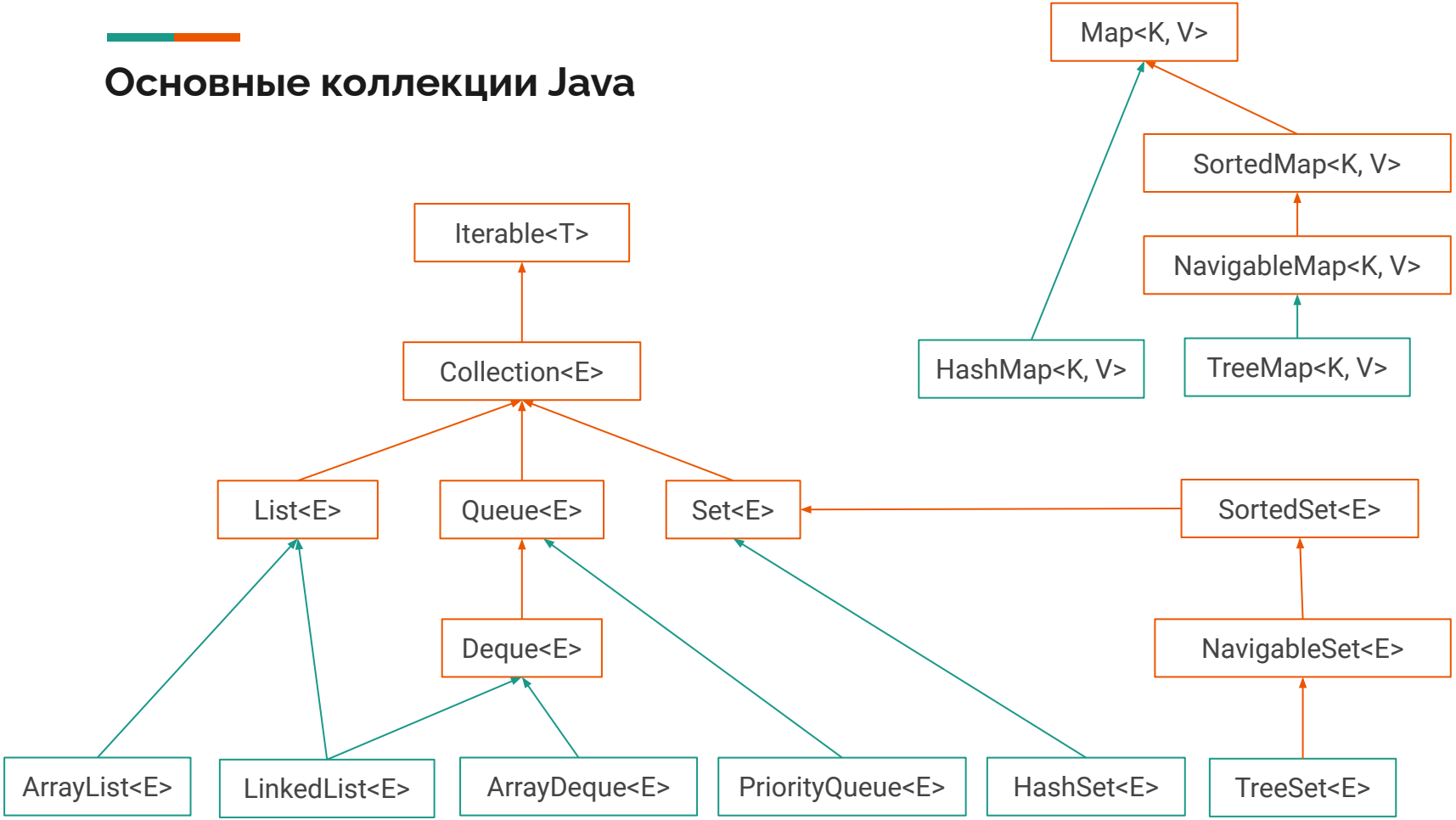
Коллекция

Набор однородных элементов, поддерживающий операции:

- вычисления размера
- перечисления всех элементов
- проверки наличия элемента
- добавления элемента
- удаления элемента

Коллекция может допускать дублирование одинаковых элементов, но может и запрещать.
Строки и массивы тоже в каком-то смысле коллекции.

Основные коллекции Java



java.util.Collection

Базовый интерфейс, который реализует большинство коллекций Java.

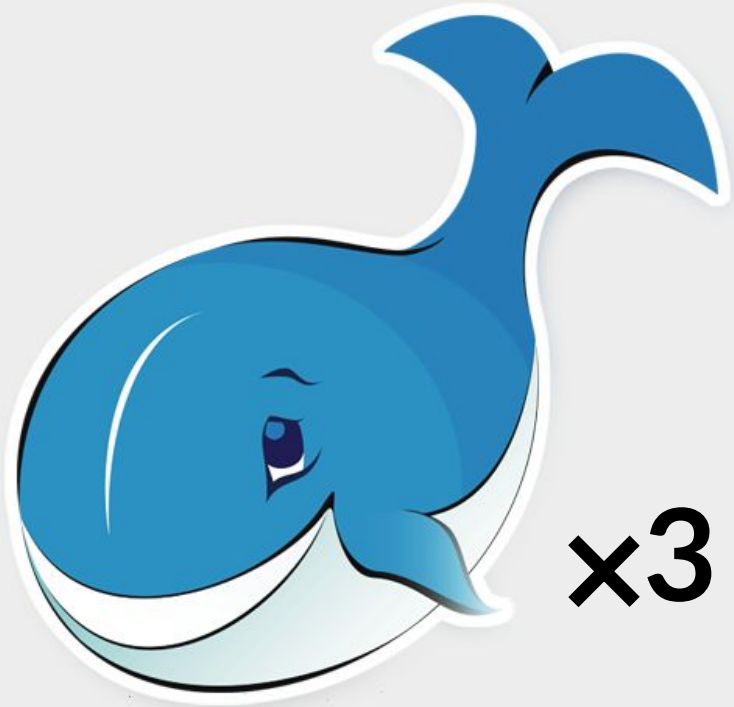
```
public interface Collection<E> extends Iterable<E> {

  int size();
  Iterator<E> iterator();
  boolean contains(Object o);
  boolean add(E e);
  boolean remove(Object o);
  void clear();
  // ...

}
```

Три кита

Список
java.util.List

Словарь
java.util.Map

Множество
java.util.Set



List

java.util.List

Упорядоченный набор значений.
Одинаковые значения могут встречаться несколько раз.
Позволяет получить элемент по порядковому номеру. А также вставить элемент в определенное место внутри списка.
Как массив, но без фиксированной длины.
Основные операции:

- операции `Collection`
- вставка элемента по индексу
- поиск элемента по индексу или значению
- изменение элемента по индексу

Основные реализации: `ArrayList`, `LinkedList`.

Создание списка и вывод его элементов

```
List<Point> points = new ArrayList<>();
points.add(new Point(1, 1));
points.add(new Point(2, 2));
points.add(new Point(1, 1));

for (Point point : points) {
  System.out.println(point);
}
```

```
Point@3e1
Point@401
Point@3e1
```

Переопределение toString

```
class Point {

  @Override
  public String toString() {
     return "(" + x + ", " + y + ")";
  }
}
```

```
(1, 1)
(2, 2)
(1, 1)
```

Элементы перечисляются в порядке их добавления.

Вставка элемента в произвольную позицию

```
List<Point> points = new ArrayList<>();
points.add(new Point(1, 1));
points.add(new Point(2, 2));
points.add(new Point(1, 1));
points.add(1, new Point(4, 4));

for (Point point : points) {
  System.out.println(point);
}
```

```
(1, 1)
(4, 4)
(2, 2)
(1, 1)
```

Вставка и удаление элементов

Все существующие элементы смещаются на одну позицию “вправо”.
Аналогично, при удалении элемента, все последующие смещаются на одну позицию “влево”.

Удаление первого элемента:
```
points.remove(0);
```

Удаление элемента по значению (первого найденного):
```
points.remove(new Point(1, 1));
```

Для поиска элементов по значению используется `equals`.

Чтение и запись значений по индексу

Первый элемент в списке:

```
Point first = points.get(0);
```

Изменение значения второго элемента (порядок остальных элементов не меняется, в отличие от метода add):

```
points.set(1, new Point(7, 1));
```

Индекс элемента

Поиск индекса элемента по значению с начала и конца списка:

```
int firstIndex = points.indexOf(new Point(4, 4)); // -1 if not found
int lastIndex = points.lastIndexOf(new Point(4, 4)); // -1 if not found
```

Различия между ArrayList и LinkedList

`ArrayList` хранит элементы в массиве, который может увеличиваться при необходимости.
`LinkedList` основан на ссылках между узлами (двусвязный список).

Зачем нужны разные реализации?

Time complexity

	ArrayList	LinkedList
`get(index)`	O(1)	O(n)
`insert(index, E)`	O(n)	O(n)
`add(E)`	O(1)..O(n)	O(1)
`it.remove()`	O(n)	O(1)

Map

java.util.Map

Словарь или ассоциативный массив. Хранит пары (ключ, значение).

Не реализует интерфейс `Collection`.

Основные операции:

- добавления значения по ключу (один ключ — одно значение)
- поиска значения по ключу
- удаления значению по ключу

Основные реализации: `HashMap`, `TreeMap`.

Добавление элементов

Пусть некоторые точки имеют буквенные обозначения.

```
Map<Point, String> labels = new HashMap<>();
labels.put(new Point(0, 0), "O");
labels.put(new Point(1, 0), "Ex");
labels.put(new Point(0, 1), "Ey");

System.out.println(labels);
```

```
{(1, 0)=Ex, (0, 0)=O, (0, 1)=Ey}
```

Получение значения по ключу

```
List<Point> points = …
Map<Point, String> labels = …
for (Point point : points) {
    String label = labels.get(point); // null, if not found
    if (label != null) {
        // show label...
    }
}
```

Если значение для заданного ключа отсутствует, возвращается `null`.

Счетчики частоты слов на основе Map

```
Map<String, Integer> wordCounts = new HashMap<>();
for (String word : words) {
  // читаем текущее значение счетчика
  Integer count = wordCounts.get(word);
  // если слово раньше не встречалось, то его нет в словаре
  if (count == null)
    count = 0;
  // увеличиваем значение счетчика
  count++;
  // записываем новое значение
  wordCounts.put(word, count);
}
```

Множество ключей и список значений

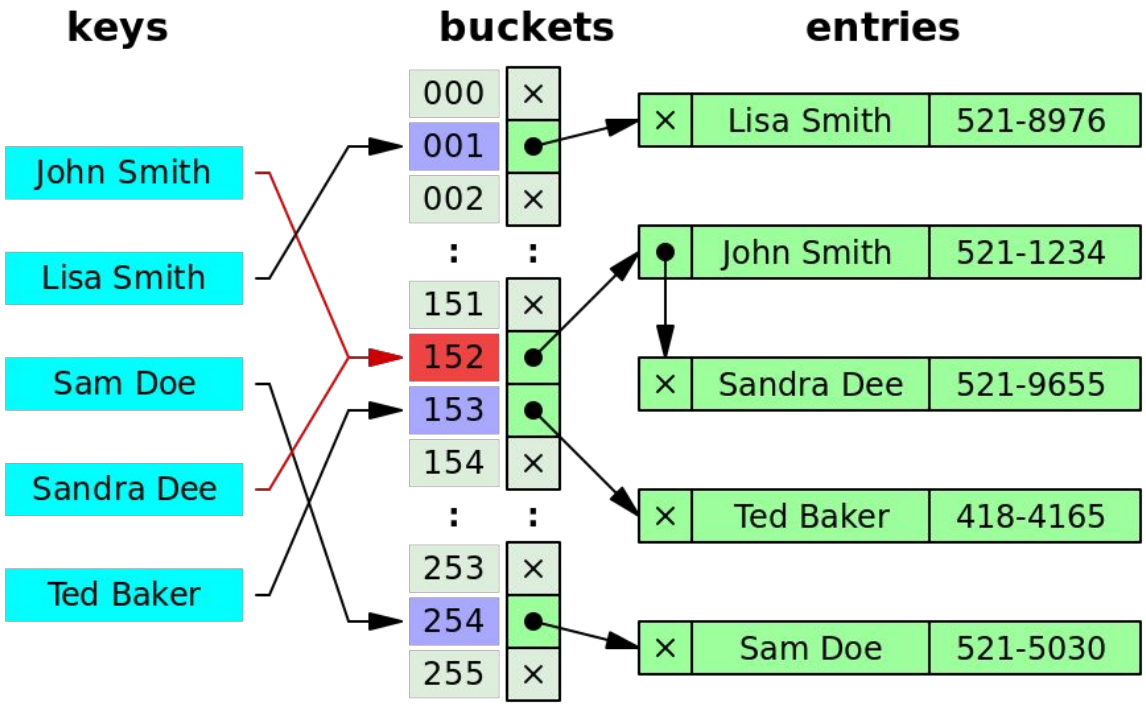
```
Map<Point, String> map = new HashMap<>();

// все ключи
Set<Point> keys = map.keySet();

// все значения
Collection<String> values = map.values();
```

Различия между HashMap и TreeMap

`HashMap` разбивает ключи на группы по хешам объектов и формирует на их основе списки адресуемых значений. Проблема: коллизии хэшей.

`TreeMap` основана на красно-черных деревьях. То есть, ключи хранятся в упорядоченном виде.

Разрешение коллизий хешей через связные списки



Time complexity

	HashMap	TreeMap
`get`	O(1)	O(lg(n))
`put`	O(1)	O(lg(n))
`remove`	O(1)	O(lg(n))

Set

java.util.Set

Неупорядоченное множество уникальных элементов. Не допускает повторения элементов.
Сравнение значений основано на `equals`.
Основные операции как у `Collection`.
Основные реализации: `HashSet`, `TreeSet`.

Подсчет уникальных слов в списке

```
List<String> words = new ArrayList<>();
words.add("day");
words.add("night");
words.add("day");

Set<String> uniqueWords = new HashSet<>();
for (String word : words) {
  uniqueWords.add(word);
}
System.out.println("Num unique words: " + uniqueWords.size());

Num unique words: 2
```

Различия между HashSet и TreeSet

`HashSet` и `TreeSet` основаны на `HashMap` и `TreeMap` соответственно.

Time complexity

	HashSet	TreeSet
`add`	O(1)	O(lg(n))
`contains`	O(1)	O(lg(n))
`remove`	O(1)	O(lg(n))