

Дженерики

1. Что такое дженерики?

Параметризованные типы для реализации обобщённого* программирования.

Начиная с 5й версии Java есть возможность параметризации **классов, интерфейсов и методов** каким-то типом. Можно объявлять и затем использовать некоторую переменную, в качестве которой будет подставлен любой тип, удовлетворяющий условиям $\langle T \rangle$.

Справка: Обобщённое программирование: особый подход к описанию данных и алгоритмов, позволяющее работать с различными типами данных без изменения их описания. Например, нужно реализовать общую библиотечную логику, которая м.б. применена к объектам разных типов.

Синтаксический сахар (англ. syntactic sugar) в языке программирования — это синтаксические возможности, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека.

2. Для чего нужны дженерики?

Неопределённость - источник ошибок. **Обобщения добавляют стабильности коду**, позволяя обнаруживать больше ошибок во время компиляции.

Типизированные параметры позволяют **повторно использовать один и тот же код с разными входными данными**.

Отличия: входными для формальных параметров являются **значения**, а входными для типизированных параметров являются **типы** (объектов класса).

Например, необходимо реализовать какую-то библиотечную логику, которая может быть применена к объектам разных типов ИЛИ реализовать метод для поиска минимального элемента в массиве. Как сделать так, чтобы для поддержки нового типа, который пользователь хочет использовать в массиве, не приходилось дублировать код, заменяя одно имя типа на другое?

Можем использовать Object на входе и на выходе (это решает проблему копипаста, но бьёт по удобству использования кода). Получается, что пользователю нужно помнить, **объекты какого типа** он хранит в массиве и аккуратно следить за тем, чтобы не положить в массив лишнего... А при извлечении объекта Object нужно приводить к необходимому типу. И ошибка проявится только во время компиляции.

Решение: вместо конкретного BigDecimal или String можно объявить и затем использовать некоторую переменную, в качестве которой будет подставлен любой тип, удовлетворяющий условиям.

```
TreeNode rootNode = new TreeNode();
rootNode.value = "foobar";
// tree manipulation

String value = (String) rootNode.value;
Object[] arrayOfBigDecimals = {...};

BigDecimal min = (BigDecimal) minElement(arrayOfBigDecimals);
```

```
public static <T extends Comparable<T>> T
minElement(T[] values) {
    if (values.length == 0) {
        return null;
    }

    T min = values[0];
    for (int i = 1; i < values.length; i++) {
        if (min.compareTo(values[i]) > 0) {
            min = values[i];
        }
    }
    return min;
}
```

Параметр vs Аргумент. (в дженериках)?

Параметр – это тип данных, которыми оперируют классы, интерфейсы указанные в дженериках. Параметр типа (type parameter) используются при объявлении дженерик-типов. Например, для Box<T> T — это ПАРАМЕТР типа.

Аргумент типа (type argument). Тип объекта, который может использоваться вместо параметра типа. Например, для Box<Paper> Paper — это АРГУМЕНТ типа.

ПРЕИМУЩЕСТВА:

Код, использующий дженерики, имеет много преимуществ по сравнению с неуниверсальным кодом:

1) Более сильная проверка типов во время компиляции

Компилятор Java применяет строгую проверку типов к универсальному коду и выдает ошибки, если код нарушает безопасность типов.

Исправить ошибки compile-time errors (времени компиляции) проще, чем исправить ошибки runtime errors (времени выполнения), которые бывает сложно найти.

2) Устранение необходимости явного приведения

Следующий фрагмент кода без дженериков требует приведения:

```
List list = new ArrayList();
list.add("Hello!");
String s = (String) list.get(0);
```

При переписывании для использования дженериков код не требует приведения:

```
List<String> list = new ArrayList<String>();
list.add("Hello!");
String s = list.get(0); // нет приведения
```

3) Возможность реализовывать общие алгоритмы.

Используя универсальные шаблоны, программисты могут реализовывать универсальные алгоритмы, которые работают с коллекциями различных типов, могут быть настроены, безопасны для типов и легче читаются.

ОГРАНИЧЕНИЯ GENERICS:**1) Параметризация возможна только ссылочным типам**

(значением параметра не могут быть примитивный тип или примитивное значение)

```
ok      TreeNode <String> stringNode;
ok      TreeNode <Integer> stringNode;
ok      TreeNode <int[]> stringNode;
не ок   TreeNode <int> stringNode;
не ок   TreeNode <10> stringNode;
```

Ограничения

- Нельзя инстанцироваться с **примитивным типами**
- Нельзя создать инстанс параметра типа
 - E e = new E();
- Статичные поля с параметром типа

```
public class MobileDevice<T> {
    private static T os;
}
```

2) Внутри параметризованного класса или метода нельзя создавать экземпляр или массив Т.**3) Также не работает проверка instance of (т.к. внутри класса ничего про конкретное E не известно)**

Нельзя создать объект Т внутри параметризованного класса T t = new T();

Type Erasure

После компиляции какая-либо информация о дженериках стирается. Это называется "**Стирание типов**"

Стирание типов и дженерики сделаны так, чтобы обеспечить обратную совместимость со старыми версиями JDK, но при этом дать возможность помогать компилятору с определением типа в новых версиях Java.

Дженерики были введены в язык Java для обеспечения более строгой проверки типов во время компиляции и для поддержки универсального программирования.

Стирание типов гарантирует, что для параметризованных типов не будут создаваться новые классы; следовательно, дженерики не несут накладных расходов во время выполнения.

<https://docs.oracle.com/javase/tutorial/java/generics/erasure.html>

2.1. Типизированные методы (Generic Methods)

Дженерики позволяют типизировать методы.

Generic methods — это методы, которые вводят собственные параметры типа.

Это похоже на объявление универсального типа, но область действия параметра типа ограничена методом, в котором он объявлен.

Допускаются статические и нестатические универсальные методы, а также конструкторы универсальных классов.

Важно запомнить про **синтаксис Generic methods**:

- включает список типизированных параметров ВНУТРИ угловых скобок <>
- список типизированных параметров идёт ДО типа возвращаемого значения метода

Статический и другой метод можно параметризовать отдельно от class, объявив generic параметры в <> ПОСЛЕ модификаторов, но ПЕРЕД именем возвращаемого значения типа.

Для статических универсальных методов раздел параметров типа должен стоять перед возвращаемым типом метода.

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue());
    }
}
```

Полный синтаксис для вызова этого метода будет следующим:

```
Pair<Integer, String> p1 = new Pair<>(1, "яблоко");
Pair<Integer, String> p2 = new Pair<>(2, "груша");
boolean same = Util.<Integer, String>compare(p1, p2);
```

Тип был указан явно, как показано жирным шрифтом. Как правило, это можно опустить, и компилятор выведет нужный тип:

```
boolean same = Util.compare(p1, p2);
```

Эта функция, известная как **type inference** (вывод типа), позволяет вызывать универсальный метод как обычный метод, не указывая тип между угловыми скобками.

2.2. Типизированные классы (Generic Types)

Типизировать можно не только методы, но и сами классы.

Если мы используем класс, ДЖЕНЕРИК указывается ПОСЛЕ имени класса.

```
public static void main(String []args) {
    SomeType<String> st = new SomeType<>();
    List<String> list = Arrays.asList("test");
    st.test(list);
}
```

Еще раз: правило #2: Если класс типизирован, всегда указывать тип в дженерике.

The most commonly used type parameter names are:

E - Element (used extensively by the Java Collections Framework)	Элемент
K - Key	Ключ
N - Number	Число
T - Type	Тип
V - Value	Значение
S,U,V etc. - 2nd, 3rd, 4th types	

При использовании шаблонов параметры типа не могут быть статическими!
Т.к. статическая переменная является общей для объекта, компилятор не может определить, какой тип использовать.

Ограничения

- Нельзя инстанцироваться с **примитивным типами**
- Нельзя создать инстанс параметра типа
- E e = new E();**

```
public class MobileDevice<T> {
    private static T os;
    // ...
}
```

Можно ли параметризовать массив?

Нет, Т.к. неизвестен тип и не понятно, сколько памяти выделить.

```
package java.util;

public final class Optional<T> {
    private final T value; объявляем поле

    private Optional(T value) { объявляем параметр метода
        this.value = Objects.requireNonNull(value);
    }

    Это T совсем другое T =) Статический/нестатический метод
    public static <T> Optional<T> of(T value) { можно объявить отдельно от класса
        return new Optional<>(value);
    }

    public T get() { объявляем тип возвращаемого значения метода
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }

    // ...
}
```

**Ограничение на тип параметра
extends ... & ... & ...**

Generic на уровне класса исп. для параметризации экземпляров класса

Полезное для понимания Optional (далее вопрос еще будет) <https://youtu.be/fbEnhHjEX3M>
На английше тот же пример (хороший канал) <https://www.youtube.com/watch?v=1xCxoOuDZuU>

Generic class может иметь несколько параметров типа.

Следующие операторы создают два экземпляра класса OrderedPair :

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");
```

Код new OrderedPair<String, Integer> создает экземпляр K как String и V как Integer .

Следовательно, типы параметров конструктора OrderedPair — String и Integer соответственно.

Из-за автоупаковки (autoboxing) допустимо передавать String и int в класс.

Также можно заменить параметр типа (то есть K или V) параметризованным типом (то есть List<String>).

Например, используя пример OrderedPair<K, V> :

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

3. Что такое сырье типы (raw type)?

Raw Types или сырье типы – это типы без указания "уточнения" параметризованного типа в угловых скобках <...>. Говоря о дженериках, мы имеем две категории: "сырые" типы (Raw Types) и типовизированные типы (Generic Types)

Сырые типы — это типы без указания "уточнения" в фигурных скобках (angle brackets):	Типовизированные типы — наоборот, с указанием "уточнения":
<p>Сырые типы все еще существуют для обеспечения обратной совместимости , поэтому компилятору необходимо различать эти <u>необработанные</u> типы и универсальные типы:</p> <pre>List generics = new ArrayList(); List raws = new ArrayList();</pre> <p>Несмотря на то, что компилятор по-прежнему позволяет нам использовать необработанные типы в конструкторе, он выдаст нам предупреждающее сообщение:</p> <p><i>ArrayList is a raw type. References to generic type ArrayList should be parameterized</i></p>	<p>Особый синтаксис, который добавили в Java SE 7, и называется он "the diamond", что в переводе означает алмаз (из-за формы фигурных скобок <>)</p> <p><i>List cars = new ArrayList<>();</i> ← <i>Box<Integer> integerBox = new Box<>();</i></p> <p>Также Diamond синтаксис связан с понятием "Type Inference", выведение типов. Компилятор, видя СПРАВА <> смотрит на левую часть, где расположено объявление типа переменной, в которую присваивается значение. И по этой части понимает, каким типом типовизировано значение справа.</p>

Даймонд оператор (всегда пустой):

Основная цель diamond оператора <> — упростить использование универсальных шаблонов при создании объекта
Это позволяет избежать непроверенных предупреждений в программе и делает программу более читаемой.
Функция вывода типов компилятора Java 1.7 определяет наиболее подходящее объявление конструктора, соответствующее вызову .

Правило #1: всегда использовать diamond синтаксис, если мы используем типовизированные типы.

В противном случае мы рискуем пропустить, где у нас используется raw type (сырой тип).

Правило #2: Если класс типовизирован, всегда указывать тип в дженерике.

*Можно ли выбрасывать исключение generic-типа?

Короткий ответ – да. Как в большинстве каверзных вопросов про дженерики, ответ становится очевидным если подумать, **во что сотрутся** типы-параметры.

Чтобы объявить, что метод выбрасывает исключение обобщенного типа T , этот тип T должен быть объявлен расширяющим Throwable .

Именно в Throwable в таком случае сотрется T при компиляции. Также в качестве типа-верхней границы можно использовать любого наследника Throwable .

```
class MyClass<T extends IOException> {
    void foo() throws T {
        // ...
    }
}
```

*Дженерики в исключениях – что можно, а что нельзя?

1. Можно выбрасывать исключение generic-типа.

Тип-параметр T может использоваться в throws , переменная типа T может использоваться в throw . См. предыдущий вопрос.

2. Нельзя использовать дженерик в catch.

Множественные блоки `catch` должны идти без повторений, в определенном порядке – от специфичного класса к более базовому. Стирание типов-параметров в связи с этими правилами добавило бы путаницу, не неся особой пользы.

3. Нельзя параметризовать класс-исключение типами.

Если вы попытаетесь скомпилировать конструкцию вида `class MyException<T> extends Throwable {}`, то увидите ошибку `generic class may not extend java.lang.Throwable`.

4. Можно реализовывать исключением generic-интерфейс.

Исключение вполне может быть, например `Comparable` или `Iterable`. Механизм обработки исключений работает на классах, никак не затрагивая интерфейсы.

4. Что такое вайлдкарды?

Wildcard — это дженерик вида `<?>`, что означает, что тип может быть чем угодно.

Используется, например, в коллекциях, где для всех коллекций базовым типом является `Collection<?>`

Представляет неизвестный тип, позволяет легко уместить **нужную логику с привязкой к конкретным типам** в один метод.

```
List<String> strings = new ArrayList<String>(); // ошибка компиляции!
```

Между объектом и коллекцией объектов есть важное различие. Если класс В является наследником класса А, то **Collection при этом — не наследник Collection<A>**.

Именно по этой причине мы НЕ можем привести `List<String>` к `List<Object>`.
`String` является наследником `Object`, но `List<String>` **не является наследником** `List<Object>`.

Знак вопроса (?) известен как wildcard (подстановка) в generic программировании.
Wildcards можно использовать как тип параметра, поля или локальной переменной,
иногда как возвращаемый тип.

В отличие от массивов, **разные экземпляры generic типа несовместимы друг с другом**, даже явно. Эту несовместимость можно смягчить с помощью wildcard если **знак вопроса используется как фактический параметр типа**.

Types of wildcards in Java:

1) Upper Bounded Wildcards (С ВЕРХНЕЙ ГРАНИЦЕЙ `<? extends A>`)

Эти Wildcards можно использовать, когда вы хотите ослабить ограничения на переменную.

Например, хотим написать метод, который работает на `List < Integer >, List < Double >, and List < Number >`

Можем реализовать, используя Wildcards с верхней границей.

Используем Wildcards ('?'), за которым следует ключевое слово `extends`, за которым следует его верхняя граница (upper bound).

```
public static void add(List<? extends Number> list)
```

2) Lower Bounded Wildcards (С НИЖНЕЙ ГРАНИЦЕЙ `<? super A>`)

Он выражается с помощью Wildcards ('?'), за которым следует ключевое слово `super`, за которым следует его нижняя граница:

```
List<? super Integer> list)
```

3. Unbounded Wildcard (НЕОГРАНИЧЕННЫЙ)

Этот тип Wildcards указывается с помощью подстановочного знака (?), например, List<?>.

Это называется **списком неизвестных типов**. Они полезны в следующих случаях —

- 1) При написании метода, который можно использовать с использованием функций, предоставляемых в классе Object.
-) Когда код использует методы универсального класса, которые не зависят от параметра типа.

```
private static void printlist(List<?> list)
```

Пример №1 Upper Bounded Wildcards (С ВЕРХНЕЙ ГРАНИЦЕЙ) <? extends A>

```
class WildcardDemo {
    public static void main(String[] args) {
        // Upper Bounded Integer List
        List<Integer> list1 = Arrays.asList(4, 5, 6, 7);
        // printing the sum of elements in list
        System.out.println("Total sum is:" + sum(list1));
        // Double list
        List<Double> list2 = Arrays.asList(4.1, 5.1, 6.1);
        // printing the sum of elements in list
        System.out.print("Total sum is:" + sum(list2));
    }

    private static double sum(List<? extends Number> list) {
        double sum = 0.0;
        for (Number i : list) {
            sum += i.doubleValue();
        }
        return sum;
    }
}
```

В приведенной программе list1 и list2 являются объектами класса List. list1 — это коллекция Integer, а list2 — это коллекция Double.

Оба они передаются методу sum, который имеет подстановочный знак, расширяющий число.

Это означает, что передаваемый список может относиться к любому полю или подклассу этого поля.

Здесь Integer и Double являются подклассами класса Number.

Пример №2 Lower Bounded Wildcards (С НИЖНЕЙ ГРАНИЦЕЙ) <? super A>

```
class WildcardDemo {
    public static void main(String[] args) {
        // Lower Bounded Integer List
        List<Integer> list1 = Arrays.asList(4, 5, 6, 7);
        // Integer list object is being passed
        printOnlyIntegerClassOrSuperClass(list1);

        // Number list
        List<Number> list2 = Arrays.asList(4, 5, 6, 7);
        // Integer list object is being passed
        printOnlyIntegerClassOrSuperClass(list2);
    }

    public static void printOnlyIntegerClassOrSuperClass(
        List<? super Integer> list)
    {
        System.out.println(list);
    }
}
```

Здесь аргументы могут быть Integer или надклассом Integer (Number).

Метод printOnlyIntegerClassOrSuperClass будет принимать только Integer или его объекты суперкласса.

Однако если мы передадим список типов Double, то получим ошибку компиляции. Это связано с тем, что можно передать только поле Integer или его суперкласс. Double не является надклассом Integer.

Пример №3 Unbounded Wildcard (НЕОГРАНИЧЕННЫЙ)

```
class Unboundedwildcarddemo {
    public static void main(String[] args) {
        // Integer List
        List<Integer> list1 = Arrays.asList(1, 2, 3);
        // Double list
        List<Double> list2 = Arrays.asList(1.1, 2.2, 3.3);
        printlist(list1);
        printlist(list2);
    }

    private static void printlist(List<?> list)
    {
        System.out.println(list);
    }
}
```

5. Расскажите про принцип PECS

Принцип PECS — Producer Extends Consumer Super.

Чтобы писать интерфейсы, абсолютно безопасные с точки зрения типов, но при этом не имеющие бессмысленных и создающих неудобства ограничений.

Этот принцип Joshua Bloch называет **PECS** (Producer Extends Consumer Super), а авторы книги Java Generics and Collections (Maurice Naftalin, Philip Wadler) — **Get And Put Principle**.

Коллекции с wildcards и ключевым словом `extends` — это producers, производители, генераторы, они лишь предоставляют данные.

Коллекции с wildcards и ключевым словом `super` — это consumers, потребители, они принимают данные, но не отдают их.

Например: `List<? extends Paper>` означает, что список может состоять из объектов типа `Paper` и всех его подтипов, а в `List<? super Paper>` могут быть объекты типа `Paper` и всех супертипов — например, `Garbage` или `Object`.

Получается, в коллекцию с `extends` нельзя добавлять, а из коллекции с `super` нельзя читать? Вроде бы всё понятно, но давайте проверим:

```
List<? super Garbage> list = new ArrayList<>();
```

Попробуем положить сюда экземпляр `Paper` — наследника `Garbage`:

```
list.add(new Paper()); // не скомпилируется
```

Получим ошибку компиляции. Ладно, тогда, может, хотя бы объект типа `Garbage` подойдёт?

```
list.add(new Garbage()); // не скомпилируется
```

И снова нет. Принцип PECS не сорвал — объект в такой список добавить нельзя. Единственное исключение — `null`. Вот так можно:

```
list.add(null); // OK
```

С первой частью принципа разобрались, теперь создадим коллекцию с ограничением снизу:

```
List<? extends Paper> list = new ArrayList<>();
```

Добавим туда один объект типа `Paper`:

```
list.add(new Paper());
```

И попробуем его же прочитать. Если верить PECS, у нас это не должно получиться:

```
list.get(0); // OK
```

Но компилятору всё нравится — никаких ошибок нет. Проблемы, впрочем, начнутся, когда мы захотим сохранить полученное значение в переменной типа `Paper` или типа, совместимого с ним:

```
Paper p = list.get(0); // не скомпилируется
```

Вторая часть принципа PECS означает, что из коллекций, ограниченных снизу, нельзя без явного приведения типа (cast) прочитать объекты граничного класса, да и всех его родителей тоже. Единственное, что доступно, — тип `Object`:

```
Object p = list.get(0); // OK
```

Тип ограничения	Что можно читать	Что можно записывать
<code><? extends SomeType></code>	Объекты <code>SomeType</code> и всех его супертипов	Только <code>null</code>
<code><? super SomeType></code>	Объекты типа <code>Object</code>	Объекты типа <code>SomeType</code> и всех его подтипов

```
List<? extends Number> list = new ArrayList<>();
list.add(1);
```

Вопрос: скомпилируются такой код?

Ответ: Нет. Только `null` может принять продюсер.

Дженерики инвариантны.

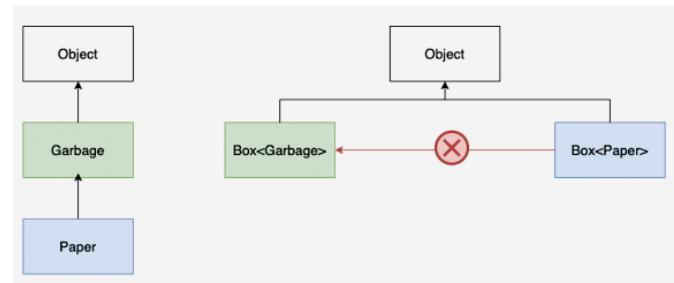
Это означает, что даже если А — это подтип В, дженерик от А НЕ является подтипов дженерика от В.

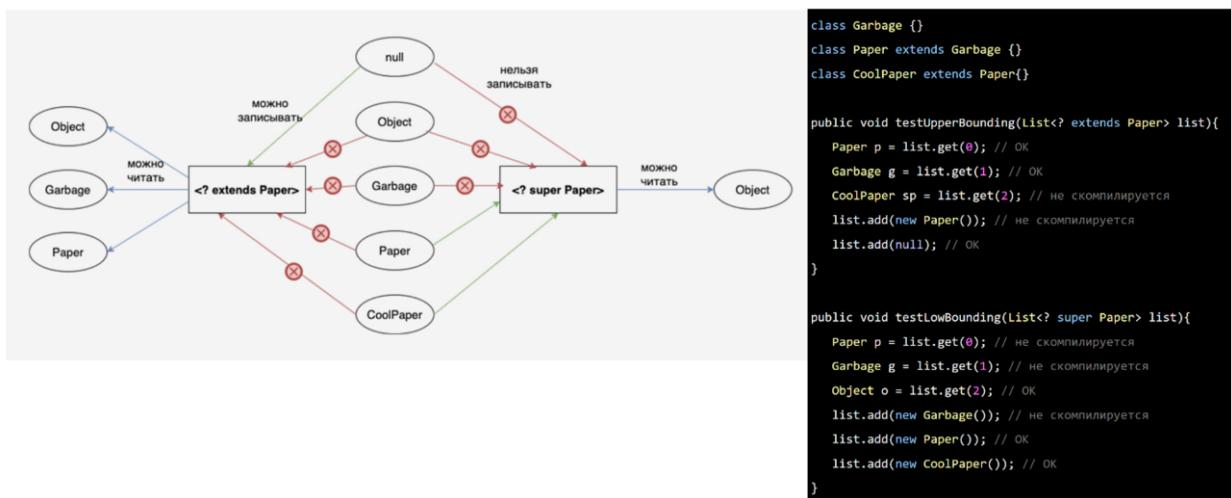
Массивы ковариантны.

Если А — это подтип В, то и `A[]` является подтипов `B[]`.

Несмотря на то, что `Paper` — это наследник `Garbage`, `Box<Paper>` — не наследник `Box<Garbage>`.

При этом оба они являются наследниками `Object`.





Статья, откуда взяты примеры кода: <https://skillbox.ru/media/code/dzheneriki-v-java-dlya-tekh-kto-postarshe/>
 Разбор и пояснения к таблице ниже <https://youtu.be/2yeFSrcTQh8>

Тип	Что можно присвоить =	Что можно читать get()	Что можно добавлять add()
Инвариантный List<Type>	только List<Type>	Type и предки Type	Type и наследники Type
Ковариантный List<? extends Type>	List<Type> и List наследников Type	Type и предки Type	Ничего
Контравариантный List<? super Type>	List<Type> и List предков Type	Object	Type и наследники Type

8. Коллекции

1. Что такое «коллекция»?

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они **имеют фиксированную длину**. Кроме того, у массивов **нет никакой защиты от изменений**.

Эту проблему в Java решают коллекции.

Коллекции – это "контейнер" или группа, т.е. **совокупность объектов**, которые мы храним вместе. Или так: коллекции — это наборы **однородных** элементов (например, страницы в книге, яблоки в корзине или люди в очереди).

Инструменты для работы с такими структурами в Java содержатся в **Java Collections Framework**, классы коллекций располагаются в пакете **java.util**

Классы коллекций являются дженериками и параметризуются типом хранимых внутри элементов. В качестве значения дженерик параметра может использоваться только ссылочный тип, поэтому все коллекции в java работают со ссылочными типами.

Если нужно хранить примитивы, то их можно превратить в объекты при помощи классов-обёрток.
А еще есть специализированные коллекции для многопоточного использования (java.util.concurrent).

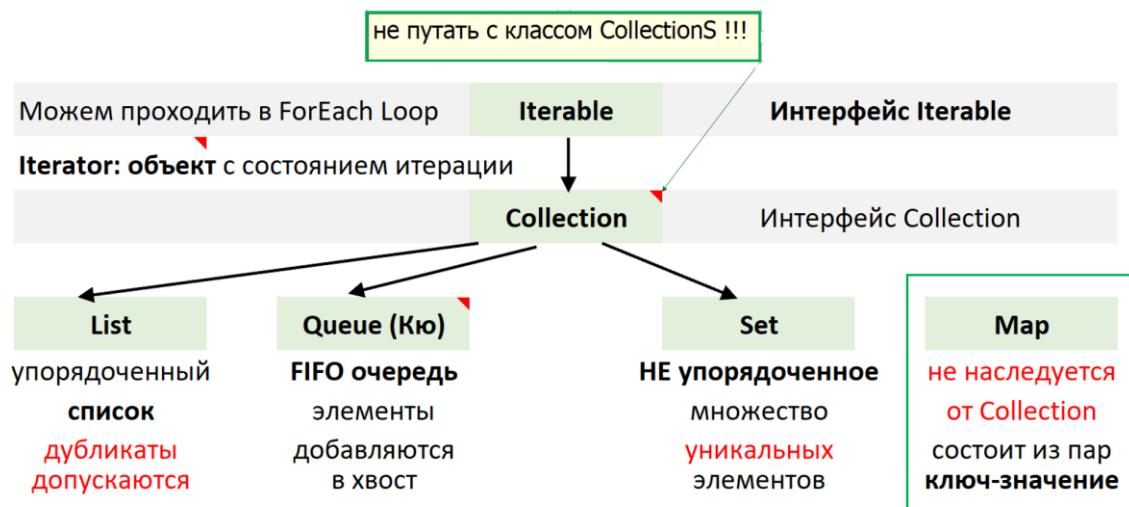
Интерфейс Collection

Интерфейс Collection является базовым для всех коллекций, определяя основной функционал:

```
1 public interface Collection<E> extends Iterable<E>{
2
3     // определения методов
4 }
```

Интерфейс Collection является обобщенным и расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в цикле по типу for-each.

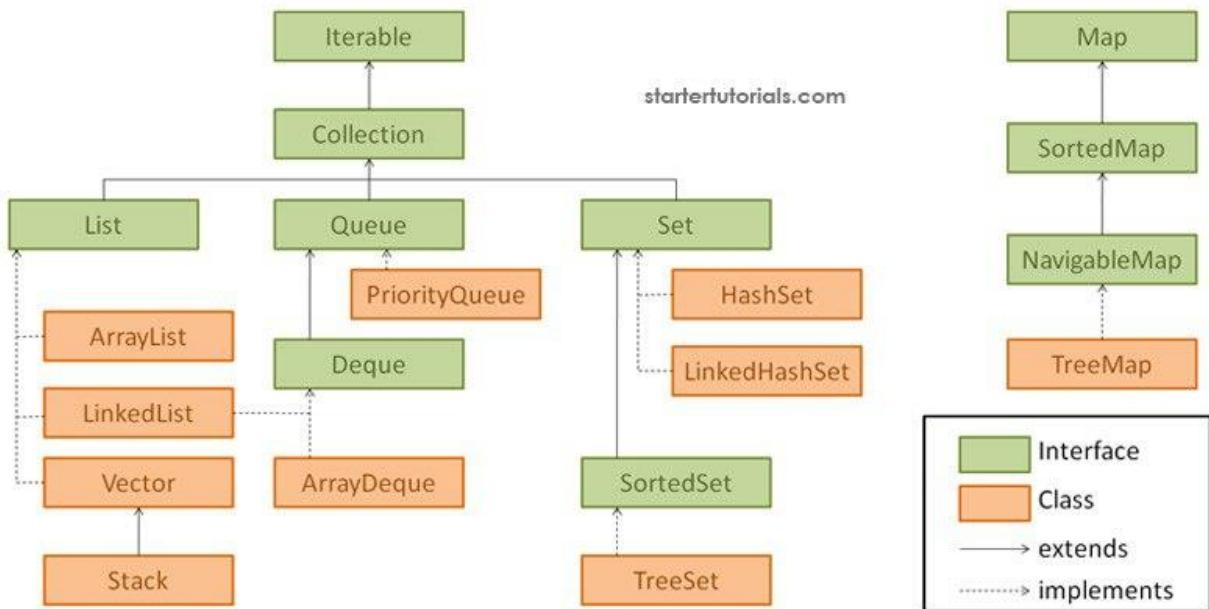
2. Расскажите про иерархию коллекций



Iterator – это объект с состоянием итерации. Он позволяет проверить, есть ли еще элементы, используя `hasNext()`, и перейти к следующему элементу (если есть), используя `next()`. Так же есть метод `remove()` – удалить во время обхода элемент коллекции.

Методы, которые есть в Collection, будут и во всех классах-наследниках (которые его имплементируют):

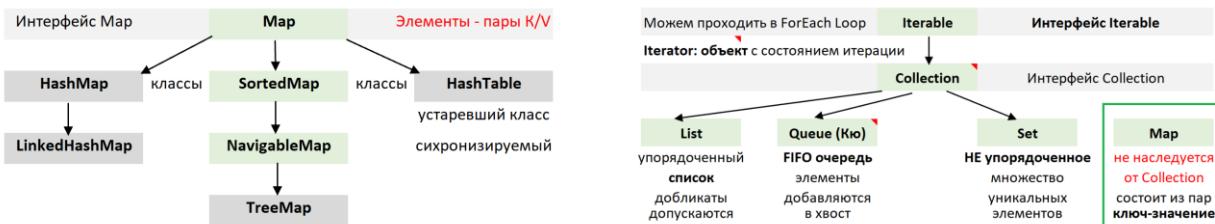
- `add()` - для добавления элементов
- `remove()` - для удаления элементов
- `size()` - проверка размера коллекции
- `isEmpty()` - проверка пуста ли коллекция или нет



3. Почему **Map** — это не **Collection**, в то время как **List** и **Set** являются **Collection**?

Интерфейс Map<K, V> представляет отображение или, иначе говоря, **словарь**, где каждый элемент представляет пару "ключ-значение" (НЕ явл. наследником **Collection**).

Такие коллекции облегчают поиск элемента, если нам известен **ключ - уникальный идентификатор объекта**.



Коллекция (**List** и **Set**) представляет собой совокупность некоторых элементов (обычно экземпляров одного класса).

Map — это совокупность пар "ключ" - "значение". Соответственно некоторые методы интерфейса **Collection** нельзя использовать в **Map**.

Например, метод `remove(Object o)` в интерфейсе **Collection** предназначен для удаления элемента, тогда как такой же метод `remove(Object key)` в интерфейсе **Map** - удаляет элемент по заданному ключу.

Карты (Map)	
key	value
1	A
2	B
3	C

4. В чем разница между классами `java.util.Collection` и `java.util.Collections`?

java.util.Collection – это **класс**, набор статических методов для работы с коллекциями.

Этот класс состоит исключительно из статических методов, которые **работают с коллекциями или возвращают их**. Он содержит полиморфные алгоритмы, которые работают с коллекциями, «обертки», которые возвращают новую коллекцию, поддерживаемую указанной коллекцией, и несколько других случаев.

Все методы этого класса вызывают исключение **NullPointerException**, если предоставленные им коллекции или объекты класса имеют значение null.

<https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>

java.util.Collection – один из основных интерфейсов Java Collections Framework.

public interface Collection<E> extends Iterable<E> это **корневой интерфейс** в иерархии коллекций. JDK не предоставляет каких-либо прямых реализаций этого интерфейса: он предоставляет реализации более конкретных под-интерфейсов, таких как Set и List.

Этот интерфейс обычно используется для передачи коллекций и управления ими там, где требуется максимальная универсальность.

Справка: Все классы реализации Collection общего назначения (которые обычно реализуют Collection косвенно через один из его под-интерфейсов) должны предоставлять **два «стандартных» конструктора**:

- конструктор void (без аргументов), который создает пустую коллекцию
- и конструктор с одним аргументом типа Коллекция, которая создает новую коллекцию с теми же элементами, что и ее аргумент.

<https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

5. Какая разница между итераторами с fail-fast и fail-safe поведением? (с примерами)

Короткий ответ: Fail-fast итератор генерирует исключение ConcurrentModificationException, если коллекция меняется во время итерации, а fail-safe – нет.

Подробнее: <https://www.youtube.com/watch?v=PILSITw4ZDc>

Это не какие-то отдельные типы, а **характеристики** разных реализаций интерфейса **Iterator**. Они определяют, как поведет себя итератор при изменении перебираемой последовательности.

Fail-fast – «БЫСТРЫЙ» итератор. Когда после его создания коллекция как-либо изменилась (добавление, изменение, удаление элемента), он падает с ошибкой без лишних разбирательств.

Так работает итератор класса **ArrayList**, при изменении он выбрасывает **ConcurrentModificationException**. Рекомендуется не основывать логику программы на fail-fast отказах, и использовать их только как признак ошибки реализации. Пример кода слева.

Fail-safe – «УМНЫЙ» итератор. Обычно плата за отказоустойчивость – возможная неконсистентность данных («слабая консистентность»).

Итератор класса **ConcurrentHashMap** **работает с копией данных**, он не выбросит исключение при изменении коллекции, но может не увидеть часть свежих изменений.

Плата за отсутствие ошибок других fail-safe итераторов может отличаться, детали всегда можно найти в документации коллекций. Пример кода ниже.

```

1b
17  private static void failFastIterator() {
18      ArrayList<String> servers = new ArrayList<String>();
19      servers.add("Tomcat");
20
21      Iterator<String> iterator = servers.iterator();
22      while (iterator.hasNext()) {
23          String server = iterator.next();
24          System.out.println(server);
25          servers.add("Jetty");
26      }
27  }
28
29
30
13
14  private static void failsafeIterator() {
15      String[] data = { "Tomcat", "Undertow", "Kubernetes", "WebSphere", "JBoss" };
16      CopyOnWriteArrayList<String> dataList = new CopyOnWriteArrayList<String>(data);
17      Iterator<String> iterator = dataList.iterator();
18      while(iterator.hasNext()) {
19          String server = iterator.next();
20          System.out.println(server);
21          if (server.equals("Tomcat")) {
22              dataList.add("Jetty");
23          }
24      }
25
26  private static void failFastIterator() {
27      ArrayList<String> servers = new ArrayList<String>();
28      servers.add("Tomcat");
29
30      Iterator<String> iterator = servers.iterator();

```

Problems Javadoc Declaration Console <terminated> FailFastAndSafe [Java Application] C:\tools\jdk11\bin\javaw.exe (Jul 19, 2021, 3:38:55 PM – 3:38:55 PM)
Tomcat
Exception in thread "main" java.util.ConcurrentModificationException
at java.base/java.util.ArrayList\$Iter.checkForComodification(ArrayList.java:1043)
at java.base/java.util.ArrayList\$Iter.next(ArrayList.java:997)
at com.mcnz.collections.FailFastAndSafe.failFastIterator(FailFastAndSafe.java:23)
at com.mcnz.collections.FailFastAndSafe.main(FailFastAndSafe.java:18)

Problems Javadoc Declaration Console <terminated> FailfastAndSafe [Java Application] C:\tools\jdk11\bin\javaw.exe (Jul 19, 2021, 3:42:12 PM – 3:42:13 PM)
Tomcat
Undertow
Kubernetes
WebSphere
JBoss

Реализация такого поведения осуществляется за счет **подсчета количества модификаций** коллекции (modification count):

- при изменении коллекции счетчик модификаций также изменяется;
- при создании итератора ему передается текущее значение счетчика;
- при каждом обращении к итератору сохраненное значение счетчика сравнивается с текущим, и, если они не совпадают, возникает исключение.

Приведите примеры итераторов, реализующих поведение fail-safe

Итератор коллекции **CopyOnWriteArrayList** (код справа) и итератор предоставления keySet коллекции **ConcurrentHashMap** являются примерами итераторов fail-safe.

Каким будет результат выполнения данного кода?

```

public static void main(String[] args) {
    List<String> stringList = new ArrayList<>();
    stringList.add("one");
    stringList.add("one and a half");
    stringList.add("two");
    stringList.add("two and a half");
    stringList.add("three and a half");

    System.out.println("Before " + stringList);
    Iterator<String> stringIterator = stringList.iterator();
    while (stringIterator.hasNext()) {
        String next = stringIterator.next();
        if (next.equals("two and a half")) {
            stringList.add("three");
        }
    }

    System.out.println("After " + stringList);
}

```

Ответ:

Before [one, one and a half, two, two and a half, three and a half]

Exception in thread "main" java.util.ConcurrentModificationException

При попытке добавить элемент в список возникнет исключение связанное с попыткой изменить список, по которому итерируемся, так как итератор для ArrayList изначально это fail-fast итератор.

В таком случае необходимо **использовать fail-safe** итераторы, они работают с клоном коллекции, которую потребовалось изменить.

В данном случае можно использовать **CopyOnWriteArrayList**.

5. Чем различаются Enumeration и Iterator?

Это два интерфейса в пакете `java.util` используются для **обхода элементов коллекции**. Хотя они выполняют одну и ту же функцию, между ними существуют некоторые различия.

Используя Enumeration, вы можете только перемещаться по коллекции, а с помощью Итератора можете также удалить элемент при обходе коллекции.

Методы – это основное различие между интерфейсами Enumeration и Iterator. Можно сказать, что Iterator – это расширенная версия Enumeration.

	Внедрено	Методы	Основные различия	fail-fast / fail-safe	Безопасность Надёжность
Enumeration	JDK 1.0.	hasMoreElements() nextElement() (Not Available)	Вы можете только перемещаться по коллекции	fail-safe (умный)	НЕТ
Iterator	JDK 1.2	hasNext() next() remove()	Вы можете также удалить элемент , перемещаясь по коллекции	fail-fast (быстрый)	ДА

Enumeration — это устаревший интерфейс, используемый для обхода только устаревших классов, таких как Vector, HashTable и Stack.

Поскольку Iterator НЕ является устаревшим, то именно он используется для обхода большинства классов в Collection Framework. Например, для обхода ArrayList, LinkedList, HashSet, LinkedHashSet, TreeSet, HashMap, LinkedHashMap, TreeMap и т. д.

Итератор является «быстрым» fail-fast итератором, то есть будет выброшено исключение ConcurrentModificationException, если коллекция изменится во время итерации с применением собственного метода remove(). Напротив, Enumeration не выбросит никаких исключений, даже если коллекция изменяется во время итерации.

Согласно Java API Docs, итератор всегда предпочтительнее перечисления:

Функции Enumeration дублируются интерфейсом Iterator. Кроме того, Iterator добавляет операцию удаления и имеет более короткие имена методов.

В новых реализациях следует рассмотреть возможность **использования Iterator вместо Enumeration**.

5.1. Как избежать ConcurrentModificationException во время перебора коллекции?

- Попробовать подобрать другой итератор, работающий по принципу fail-safe. К примеру, для List можно использовать **ListIterator**.
- Использовать ConcurrentHashMap и CopyOnWriteArrayList
- Преобразовать список в массив и перебирать массив
- Блокировать изменения списка на время перебора с помощью блока synchronized

Отрицательная сторона последних двух вариантов – это ухудшение производительности

6. Как между собой связаны Iterable, Iterator и «for-each»?

Interface Iterable<T> – это интерфейс, который реализуют коллекции, у него есть метод iterator<T>, который возвращает объект Итератор по элементам типа T.

Iterator — объект с состоянием итерации (бегунок, курсор, указатель).

Он позволяет проверить, есть ли еще больше элементов, используя hasNext(), и перейти к следующему элементу (если есть), используя next(). Так же есть метод remove().

forEach — это разновидность цикла for, **метод интерфейса Iterable**. Выполняет заданное действие для каждого элемента Iterable до тех пор, пока все элементы не будут обработаны или действие не вызовет исключение.

```
default void forEach(Consumer<? Super T> action)
```

```
1 List <Integer> numbers = new ArrayList<>(Arrays.asList(1,2,3,4,5,6,7));
2 numbers.forEach(s -> System.out.print(s + " "));
```

```
1 2 3 4 5 6 7
```

The default implementation behaves as if:

```
for (T t : this)
    action.accept(t);
```

8. Можно ли, обходя ArrayList, удалить элемент? Какое выбросит исключение?

Да, можно, но... единственный способ удалить элемент из коллекции при обходе, не получив при этом **ConcurrentModificationException** или неопределенное поведение – удалить с помощью `remove()` того же инстанса итератора.

```
Iterator<Cat> catIterator = cats.iterator(); //создаем итератор
while(catIterator.hasNext()) { //до тех пор, пока в списке есть элементы

    Cat nextCat = catIterator.next(); //получаем следующий элемент
    if (nextCat.name.equals("Филипп Маркович")) {
        catIterator.remove(); //удаляем кота с нужным именем
    }
}

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

8.1. Как удалить элемент из ArrayList при итерации?

Обычно формулируется в виде задачи на внимательность «**что здесь не так**», например →

```
for (String item : arrayList)
    if (item.length() > 2)
        arrayList.remove(item);
```

Подвох в том, что итератор `ArrayList`, который используется в таком варианте цикла `for`, является fail-fast, то есть не поддерживает итерацию с параллельной модификацией. А параллельная модификация случается даже в одном потоке, что демонстрирует этот пример. Следующий шаг итератора после удаления элемента выбросит `ConcurrentModificationException`.

Не исключение, но неожиданный результат получится если пользоваться не итератором, а обычным циклом `for` – при каждом удалении нумерация элементов будет сдвигаться.

Единственный способ удалить элемент из коллекции при обходе, не получив при этом `ConcurrentModificationException` или неопределенное поведение – удалить с помощью `remove()` того же инстанса итератора. Вариант `ListIterator` поможет, если в теле цикла требуется и работа с индексами.

Некоторые коллекции (`CopyOnWriteArrayList` и `ConcurrentHashMap`), адаптированные под многопоточную среду и имеют fail-safe итераторы.

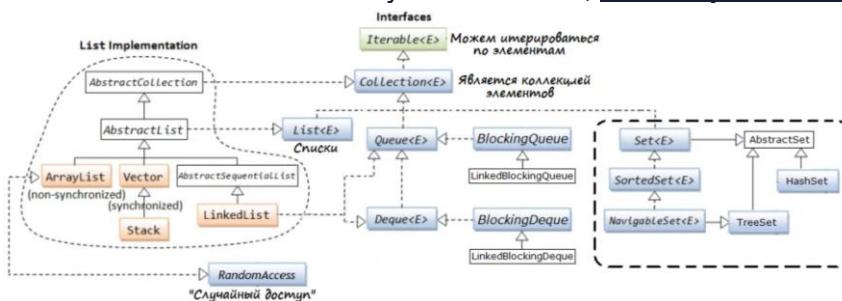
9. Как поведёт себя коллекция, если вызвать `iterator.remove()`?

Если вызову `iterator.remove()` предшествовал вызов `iterator.next()`, то `iterator.remove()` удалит элемент коллекции, на который указывает итератор, в противном случае будет выброшено `IllegalStateException`.

10. Чем Set отличается от List?

`List` – упорядоченный список с возможностью содержания дубликатов и доступа по индексу (random access, произвольно получить элемент по индексу).

`Set` – не обязательно упорядоченное множество уникальных (с точки зрения `equals`) значений. Чтобы из `Set` получить элемент, используется Iterator.

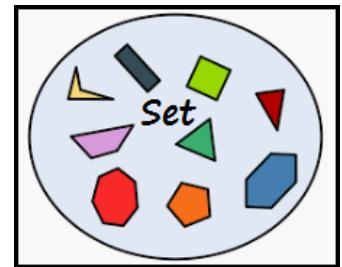


11. Расскажите про интерфейс Set.

Set — переводится как "множество". От очереди и списка Set отличается большей абстракцией над хранением элементов.

Set — как мешок с **уникальными предметами**, где неизвестно, как лежат предметы и в каком порядке они легли.

В Java такой набор представлен интерфейсом [java.util.Set](#)



Set — "collection that contains **NO duplicate** elements".

Интересно, что сам интерфейс Set не добавляет новых методов к **интерфейсу Collection**, а лишь **уточняет требования** (про отсутствие дубликатов).

В основе Set лежит Map, у которого:

- ключи — это элементы множества Set
- значения — это константа-заглушка

```
// Объект класса Object, каждый раз выступающий в роли значения в Map
private static final Object PRESENT = new Object();
```

Почему так? Так как Map хранит пары ключ-значения, а набору Set нужны только значения, то элемент Set занимает место ключа, а место значения Java заполняет пустым объектом (new Object).

Список (List) хранит все элементы, которые в него добавили. Множество (Set) хранит только **уникальные элементы**. Повторяющиеся элементы просто не добавляются во множество.



```
Set<Integer> intSet = new HashSet<>(Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21));
System.out.println(intSet);
// [1, 2, 3, 5, 21, 8, 13]
```

Метод add у множеств возвращает true, если элемент был добавлен, и false, если он уже есть во множестве.

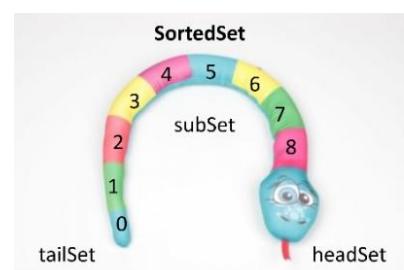
```
System.out.println(intSet.add(21)); // false
System.out.println(intSet.add(34)); // true
System.out.println(intSet.add(34)); // false
```

12. Расскажите про реализации интерфейса Set

Set имеет несколько связанных с собой интерфейсов.

Реализации **SortedSet** дают линейный порядок множества. Элементы **упорядочены по возрастанию**.

Порядок либо **натуральный** (элементы реализуют интерфейс Comparable), либо его определяет **переданный в конструктор Comparator**.



Этот интерфейс добавляет методы получения подмножества от указанного элемента (tailSet), до элемента (headSet), и между двумя (subSet). Подмножество включает нижнюю границу, не включает верхнюю.

SortedSet расширяется интерфейсом NavigableSet для итерации по порядку, получения ближайшего снизу (floor), сверху (ceiling), большего (higher) и меньшего (lower) заданному элемента.

SortedSet расширен методами навигации, сообщающими о ближайших совпадениях для заданных целей поиска.

NavigableSet добавляет к привычному iterator (который идёт от меньшего к большему) итератор для обратного порядка — **descendingIterator**.

Кроме того, NavigableSet позволяет при помощи метода **descendingSet** получить вид на себя (View), в котором элементы идут в обратном порядке (через полученный элемент можно изменять элементы изначального Set).

Интересно, что NavigableSet, подобно Queue, умеет получать и убирать элементы из набора: **pollFirst** (минимальный) и **pollLast** (максимальный).

Какие еще есть реализации?

- на основе хэш-кода — **HashSet**
- двусвязного списка — **LinkedHashSet**
- на основе красно-чёрного дерева — **TreeSet**

12.1. Для множеств ENUM-ов есть специальный класс **java.util.EnumSet**? Зачем? Чем авторов не устраивал **HashSet** или **TreeSet**?

EnumSet — это одна из разновидностей реализаций интерфейса Set для использования с перечислениями (Enum). EnumSet использует массив битов для хранения значений (bit vector), что позволяет получить высокую компактность и эффективность.

В структуре данных хранятся объекты только одного типа Enum, который указывается при создании экземпляра EnumSet. Все основные операции выполняются за константное время O(1) и в основном несколько быстрее (хотя и не гарантированно), чем их аналоги в реализации HashSet.

Пакетные операции (bulk operations, например, **containsAll()** и **retainAll()**) выполняются очень быстро, если их аргументом является экземпляр типа Enum.

Помимо этого, класс EnumSet предоставляет множество статических методов инициализации для упрощенного и удобного создания экземпляров.

Итерация по EnumSet осуществляется согласно порядку объявления элементов перечисления.

12.1. Зачем нужны и чем отличаются интерфейсы Comparable и Comparator?

```
1 class Person implements Comparable<Person>{  
2  
3     private String name;  
4     Person(String name){  
5         this.name = name;  
6     }  
7     String getName(){return name;}  
8  
9     public int compareTo(Person p){  
10         return name.compareTo(p.getName());  
11     }  
12 }  
13  
14 }
```

Интерфейс Comparable (сравнимый) содержит один единственный метод **int compareTo(E 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 возвратит ноль, значит, оба объекта равны.

Интерфейс Comparator содержит ряд методов, ключевым из которых является метод **compare()**, который возвращает числовое значение - если оно отрицательное, то объект **a** предшествует объекту **b**, иначе - наоборот. А если метод возвращает ноль, то объекты равны.

Для применения интерфейса нам сначала надо **создать класс компаратора**, который реализует этот интерфейс.

Например, сортируем дома по цене →

Создадим объект класса **PriceComparator**, а потом вызовем у **ArrayList** метод **sort()**, который принимает на вход объект класса, реализующего интерфейс **Comparator** и отсортируем **ArrayList**.

В консоли получим:

```
Sorted:
Area: 40, price: 70000, city: Oxford, hasFurniture: true
Area: 100, price: 120000, city: Tokyo, hasFurniture: true
Area: 70, price: 180000, city: Paris, hasFurniture: false
Process finished with exit code 0
```

РАЗНИЦА:

Comparable реализуется ВНУТРИ класса. По сути, определяет обычный/естественный порядок сравнения объектов. **ОДИН** метод **compareTo()**;

Comparator - реализуется ВНЕ класса.

Можно реализовать разные варианты сортировки, основанные на **сравнении различных полей** (свойств объектов). Имеет **РЯД** методов, ключевой их них – **compare()**;

13. В чем отличия TreeSet и HashSet?

Класс HashSet реализует интерфейс Set, основан на **хэш-таблице** (оптимизирован для быстрого поиска). Не запоминает порядок добавления элементов. Может хранить null.

Класс TreeSet реализует интерфейс SortedSet реализован на основе **красно-чёрного дерева** (хранит элементы в отсортированном по возрастанию порядке).

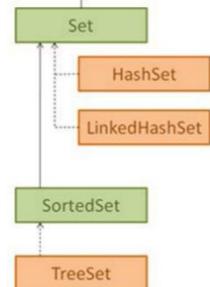
14. Чем LinkedHashSet отличается от HashSet?

LinkedHashSet отличается от **HashSet** только тем, что в его основе лежит **LinkedHashMap** вместо **HashMap**. Благодаря этому порядок элементов при обходе коллекции является идентичным порядку добавления элементов (*insertion-order*, то есть **хранит элементы в порядке добавления**).

При добавлении элемента, который уже присутствует в **LinkedHashSet** (т.е. с одинаковым ключом), порядок обхода элементов не изменяется.

```
1 public class PriceComparator implements Comparator<House> {
2
3     public int compare(House h1, House h2) {
4         if (h1.price == h2.price) {
5             return 0;
6         }
7         if (h1.price > h2.price) {
8             return 1;
9         }
10        else {
11            return -1;
12        }
13    }
14 }
```

```
3     public static void main(String[] args) {
4
5         ArrayList<House> myHouseArrayList = new ArrayList<House>();
6
7         House firstHouse = new House(100, 120000, "Tokyo", true);
8         House secondHouse = new House(40, 70000, "Oxford", true);
9         House thirdHouse = new House(70, 180000, "Paris", false);
10
11        myHouseArrayList.add(firstHouse);
12        myHouseArrayList.add(secondHouse);
13        myHouseArrayList.add(thirdHouse);
14
15        for (House h: myHouseArrayList) {
16            System.out.println(h);
17        }
18
19        PriceComparator myPriceComparator = new PriceComparator();
20        myHouseArrayList.sort(myPriceComparator);
21
22        System.out.println("Sorted: ");
23        for (House h: myHouseArrayList) {
24            System.out.println(h);
25        }
26    }
27 }
```



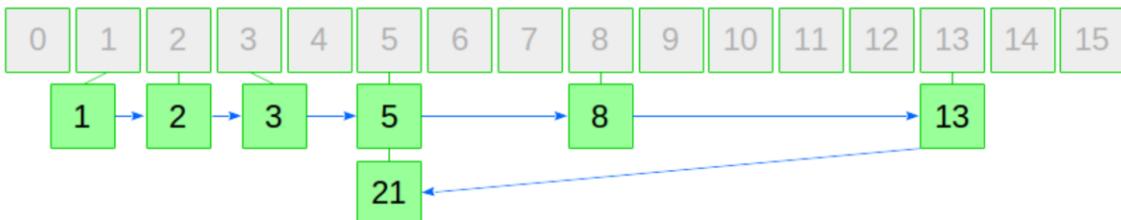
Порядок элементов в наборе Elements order

- **HashSet** - порядок не гарантируется
- **TreeSet** - порядок гарантируется сортировкой
- **LinkedHashSet** - порядок согласно очереди вставки элементов

Если нужно сохранить тот порядок, в котором элементы были добавлены во множество, можно использовать `LinkedHashSet` — реализацию `Set`, которая хранит элементы в том порядке, в котором они были добавлены.



```
Set<Integer> intSet = new LinkedHashSet<>(Arrays.asList(1, 1, 2, 3, 5, 8, 13, 21));
System.out.println(intSet);
// [1, 2, 3, 5, 8, 13, 21]
```



Хеш-таблица (Hash table) и связанный список (linked list) имплементируют интерфейс `Set` с **предсказуемым порядком итерации**. `LinkedHashSet` отличается от `HashSet` тем, что **поддерживает двусвязный список**, проходящий через все его записи.

Этот связанный список определяет порядок итерации, то есть порядок, в котором элементы были вставлены в набор (порядок вставки, insertion-order).

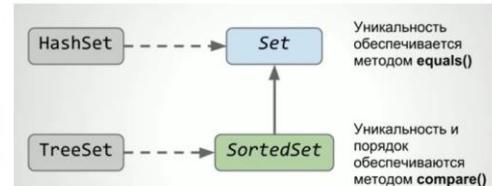
Обратите внимание, что порядок вставки не изменяется, если элемент повторно вставляется в набор. Элемент `e` повторно вставляется в множество `s`, если `s.add(e)` вызывается, когда `s.contains(e)` возвращает значение `true` непосредственно перед вызовом).

15. Что будет, если добавлять элементы в TreeSet по возрастанию?

В основе `TreeSet` лежит красно-черное дерево, которое умеет само себя балансировать. В итоге, `TreeSet` все равно в каком порядке вы добавляете в него элементы, преимущества этой структуры данных будут сохраняться.

15.1. Что будет в этом случае?

```
Set<Integer> set = new TreeSet<>(new Comparator<Integer>() {
    public int compare(Integer o1, Integer o2) {
        return 0;
    }
});
set.add(1); set.add(2); set.add(1);
System.out.println(set);
```



Подсказка: как обеспечивается уникальность элементов в `TreeSet`? Метод `compare()`
Ответ: получаем один элемент списка (проверить в идее).

16. Как устроен HashSet, сложность основных операций.

В основе HashSet лежит HashMap, у которого:

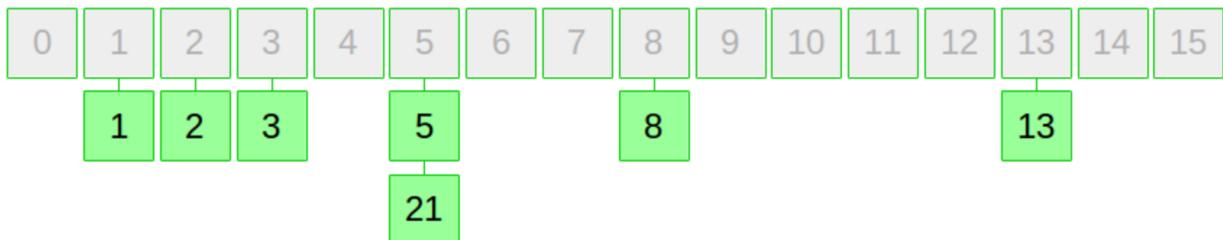
- ключи – это элементы `HashSet`
- значения – это константа-заглушка

```
// Объект класса Object, каждый раз выступающий в роли значения в Map
private static final Object PRESENT = new Object();
```

`HashSet` хранит элементы таким образом, чтобы элемент можно было очень быстро найти. Метод `contains()` у `HashSet` ищет элемент быстро, так как под капотом `HashMap`:

элементы находятся в так называемых корзинах/бакетах/entry, которые выбираются исходя из значений самих элементов(поиск бакета по хэшкоду, см. далее HashMap).

На рисунке 16 серых корзин/бакетов/узлов/нод и 7 зелёных элементов множества.



Операции добавления, удаления и поиска будут выполняться за **константное время $O(1)$** (то есть быстро) при условии, что хэш-функция правильно распределяет элементы по «корзинам». Для получения элементов из Set используется Iterator, операции быстрые.

Внимание, у Set нет метода get()! – часто спрашивают =)

	Временная сложность							
	Среднее				Худшее			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))

Несколько важных пунктов о HashSet:

- Класс реализует интерфейс Set, он может хранить только уникальные значения
- Может хранить NULL – значения
- Порядок добавления элементов вычисляется с помощью хэш-кода
- HashSet также реализует интерфейсы Serializable и Cloneable

Для поддержания постоянного времени выполнения операций время, затрачиваемое на действия с HashSet, должно быть **прямо пропорционально количеству элементов в HashSet + «емкость» встроенного экземпляра HashMap** (количество «корзин»).

Поэтому для поддержания производительности очень важно не устанавливать слишком высокую начальную ёмкость (или слишком низкий коэффициент загрузки).

17. Как устроен LinkedHashSet, сложность основных операций.

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

18. Как устроен TreeSet, сложность основных операций.

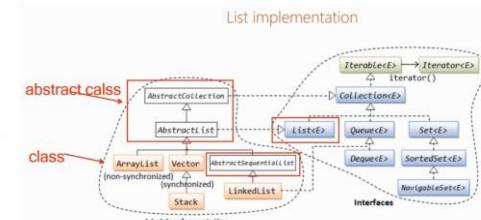
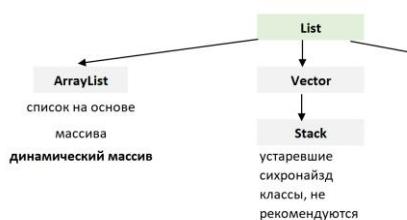
В основе TreeSet лежит TreeMap, у которого:

- ключи – это элементы TreeSet
- значения – это константа-заглушка

HashSet не может гарантировать, что данные будут отсортированы, так как работает по другому алгоритму. Если сортировка для вас важна, то используйте **TreeSet**.

19. Расскажите про интерфейс List

List – список или упорядоченная последовательность элементов, позволяющая хранить **дубликаты и null**. Каждый элемент имеет индекс (от нуля и дальше), поиск по индексу.



```

public void onClick(View view) {
    SortedSet<String> countrySet = new TreeSet<>();
    countrySet.add("Россия");
    countrySet.add("Франция");
    countrySet.add("Гондурас");
    countrySet.add("Кот-Д'Ивуар"); // любимая страна всех котов

    mInfoTextView.setText(countrySet.toString());
}
  
```

Названия стран выводятся в алфавитном порядке.

20. Как устроен ArrayList, сложность основных операций.

ArrayList реализует интерфейс List.

ArrayList – это динамический массив, т.е. может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта. Элементы ArrayList могут быть абсолютно любых типов в том числе и null.

Используем тогда, когда нам нужна структура, похожая на массив, но где нам нужно добавлять/удалять/изменять элементы. Получение и изменение элементов **выполняется быстро**, поскольку эти операции просто обращаются к соответствующему элементу массива. В основе ArrayList лежит массив Object (элементами явл. Объекты типа Object).

Ёмкость **capacity** массива по дефолту – **10 мест** (не путать размер и ёмкость).

Размер массива – это сколько по факту лежит элементов в массиве, а ёмкость – это потенциально возможное кол-во мест).

Создание:

- 1) `ArrayList<DataType> list1 = new ArrayList<DataType>();`
- 2) `ArrayList<DataType> list1 = new ArrayList<>();` можем не писать повторно

Ёмкость capacity (по дефолту = 10 мест)
Если ёмкость знаем заранее, то нужно это явно указать

- 3) `ArrayList<DataType> list2 = new ArrayList<>(int capacity: 200); // сделали так - увеличили скорость добавления элементов`
Это размер базового массива, который используется для хранения элементов.
Ёмкость растет автоматически по мере добавления элементов в список массивов.
Ёмкость - это размер базового массива, который используется для хранения элементов.

НЕ ПУТАТЬ: size (это размер, т.е. сколько фактически лежит внутри элементов) и capacity (ёмкость, т.е. сколько мест выделено под заполнение)

- 4) `ArrayList<DataType> list4 = new ArrayList<>[list3]; // в параметры конструктора помещаем другой AL // ЭТО НОВЫЙ ОБЪЕКТ`

Конструктор и описание

<code>ArrayList()</code>	конструктор создает пустой список массивов.
<code>ArrayList(Collection c)</code>	конструктор строит список массивов, кот. иниц-ся элем. коллекции с.
<code>ArrayList(int capacity)</code>	конструктор создает список массивов с указанной начальной capacity.

Скорость основных операций

	Временная сложность							
	Среднее				Худшее			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)

- Быстрый доступ к элементам по индексу за константное 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

Есть понятие Время доступа (O большая). В LinkedList (связанный список) идет перебор элементов, поэтому при вставке в конец мы сначала проходим весь список (линейное время), а потом добавляем массив. Линейное время. А ArrayList - это динамический массив (доступ к элементам по индексу). Поиск значения O(n), а в ArrayList O(1) то есть константное время в любую точку массива вставка.

ОТВЕТ: в середину и в конец динам. массива ArrayList быстрее вставка (еще нативный метод используется `System.arraycopy()`);
А LinkedList используется редко, т.к. он много места в памяти занимает и подходит для небольшого кол-ва данных.

ВОТ СЛУЧАЙ, ГДЕ LinkedList быстрее, в остальном ArrayList - чемпион!

--Insert elements to begin(100k) --- Add elements (6kk) / Remove elements from begin (100k) / Set elements at end (3kk)
 LinkedList: 132 ms _____ / LinkedList: 2264 ms _____ / LinkedList: 2 ms _____ / LinkedList: 40 ms
 ArrayList: 2742 ms _____ / ArrayList: 493 ms _____ / ArrayList: 3220 ms _____ / ArrayList: 267 ms
 LinkedList is faster _____ / **ArrayList is faster** _____ / **LinkedList is faster** _____ / **LinkedList faster**

Алгоритм основных операций <https://habr.com/ru/post/128269/>

Если вызывается конструктор без параметров, то по умолчанию будет создан массив из 10-ти элементов типа Object (с приведением к типу, разумеется), индексы от 0 до 9.

`elementData = (E[]) new Object[10];`

Можно использовать конструктор **ArrayList(capacity)** и указать свою начальную емкость.

- 1) **Добавление элементов** `list.add("0");`

Внутри метода **add(value)** происходят следующие вещи:

- проверяется, достаточно ли места в массиве для вставки нового элемента `ensureCapacity(size + 1);`
- добавляется элемент в конец (согласно значению **size**) массива `elementData[size++] = element;`

Если места в массиве недостаточно, новая емкость рассчитывается по формуле **(oldCapacity * 3) / 2 + 1**.

```
// newCapacity - новое значение емкости
elementData = (E[])new Object[newCapacity];
```

Второй момент — это копирование элементов. Оно осуществляется с помощью нативного метода **System.arraycopy()**, который написан не на языке Java.

```
// oldData - временное хранилище текущего массива с данными
System.arraycopy(oldData, 0, elementData, 0, size);
```

При добавлении 11-го элемента, проверка показывает что места в массиве нет. Соответственно создается новый массив и вызывается **System.arraycopy()**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
"0"	"1"	"2"	"3"	"4"	"5"	"6"	"7"	"8"	"9"	null	null	null	null	null	null

После этого добавление элементов продолжается

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
"0"	"1"	"2"	"3"	"4"	"5"	"6"	"7"	"8"	"9"	"10"	null	null	null	null	null

- 2) **Добавление в «середину» списка** `list.add(5, "100");`

Добавление элемента на позицию с определенным индексом происходит в три этапа:

- проверяется, достаточно ли места в массиве для вставки нового элемента `ensureCapacity(size+1);`
- подготавливается место для нового элемента с помощью **System.arraycopy();** `System.arraycopy(elementData, index, elementData, index + 1, size - index);`
- перезаписывается значение у элемента с указанным индексом `elementData[index] = element;` `size++;`

В случаях, когда происходит вставка элемента по индексу и при этом в вашем массиве нет свободных мест, то вызов **System.arraycopy()** случится дважды: первый в **ensureCapacity()**, второй в самом методе **add(index, value)**, что явно скажется на скорости всей операции добавления.

В случаях, когда в исходный список необходимо добавить другую коллекцию, да еще и в «середину», стоит использовать метод **addAll(index, Collection)**. И хотя, данный метод скорее всего вызовет **System.arraycopy()** три раза, в итоге это будет гораздо быстрее поэлементного добавления.

3) Удаление элементов

Удалять элементы можно двумя способами:

- по индексу **remove(index)**
- по значению **remove(value)**

С удалением элемента по индексу всё достаточно просто: `list.remove(5);`

- Сначала определяется, какое количество элементов надо скопировать
`int numMoved = size - index - 1;`
- затем копируем элементы используя **System.arraycopy()**
`System.arraycopy(elementData, index + 1, elementData, index, numMoved);`
- уменьшаем размер массива и забываем про последний элемент
`elementData[--size] = null; // Let gc do its work`

При удалении по значению, в цикле просматриваются все элементы списка, до тех пор, пока не будет найдено соответствие. Удален будет лишь первый найденный элемент.

При удалении элементов текущая величина capacity не уменьшается, что может привести к своеобразным утечкам памяти. Поэтому не стоит пренебрегать методом **trimToSize()**.

21. Как устроен **LinkedList**, сложность основных операций. <https://habr.com/ru/post/337558/>
22. Почему **LinkedList** реализует и **List**, и **Deque**?

LinkedList – список связанных элементов, каждый из которых хранит ссылки на следующий и предыдущий элементы в списке, цепочке. Бывает односвязный и двусвязный списки.

ВАЖНО! В каком порядке элементы добавлены, в таком они и находятся.

LinkedList — класс, реализующий два интерфейса — **List** и **Deque**.

Это обеспечивает возможность создания двунаправленной очереди из любых (в том числе и null) элементов. **Каждый объект**, помещенный в связанный список, **является узлом/нодом**. Каждый узел содержит элемент, ссылку на предыдущий и следующий узел. Фактически связанный список состоит из последовательности узлов, каждый из которых предназначен для хранения объекта определенного при создании типа.

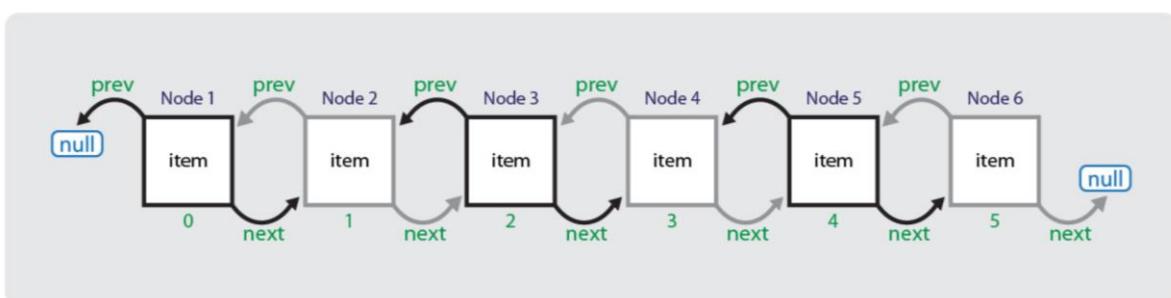


Рис. 1: Общий вид связанного списка

Каждый элемент "знает" своих соседей и ссылается на них. **И знает ТОЛЬКО соседей.**

Чтобы вставить элемент в середину такого списка, достаточно изменить ссылки на его будущих соседей.

Но чтобы получить элемент № 130, **нам нужно перебрать каждый элемент от 0 до 130.**

Операции получения и изменения выполняются медленно !

Если следующего элемента нет, то предыдущий элемент имеет ссылку на null и так LL понимает, что это конец. Бывают двух видов: DOUBLY (элементы ссылаются на 2x соседей) и SINGLY (1 ссылка на следующего)

Алгоритм основных операций.

1) Создание связанного списка

Данный код создает объект класса **LinkedList** и сохраняет его в ссылке numbers. Созданный объект предназначен для хранения целых чисел (**Integer**). Пока этот объект пуст.

Класс **LinkedList** содержит три поля →

// модификатор transient указывает на то, что данное свойство класса нельзя сериализовать

```
transient int size = 0;
transient Node<E> first;
transient Node<E> last;
```

size	0
first	null
last	null

Рис. 2: Состояние объекта сразу после создания

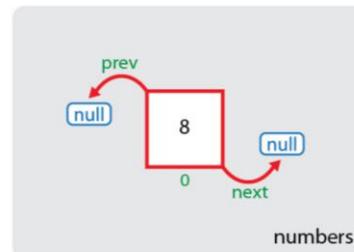
2) Добавление объекта в конец связанного списка

Данный код (`numbers.add(8)`) добавляет число 8 в конец ранее созданного списка.

Под «капотом» этот метод вызывает ряд других методов, обеспечивающих создание объекта типа **Integer**, создание нового узла, установку объекта класса **Integer** в поле item этого узла, добавление узла в конец списка и установку ссылок на соседние узлы.

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```



size	1
first	
last	

Рис. 3: Добавление первого объекта в связанный список

Для установки ссылок на предыдущий и следующий элементы **LinkedList** использует объекты своего вложенного класса **Node**.

При каждом добавлении объекта в список создается один новый узел, а также изменяются значения полей связанных списков (`size`, `first`, `last`).

В случае с добавлением первого элемента создается узел, у которого предыдущий и следующий элементы отсутствуют, т.е. являются `null`, размер коллекции увеличивается на 1, а созданный узел устанавливается как первый и последний элемент коллекции.

Добавим еще один элемент в нашу коллекцию: `numbers.add(5)`

Сначала создается узел для нового элемента (число 5) и устанавливается ссылка на существующий элемент (узел с числом 8) коллекции как на предыдущий, а следующим элементом у созданного узла остается `null`. Также этот новый узел сохраняется в переменную связанных списков `last`.

Как можно увидеть на рис. 4, первый элемент коллекции (под индексом 0) пока ссылается на `null` как на следующий элемент. Теперь эта ссылка заменяется и первый элемент начинает ссылаться на второй элемент коллекции (под индексом 1), а также увеличивается размер коллекции:

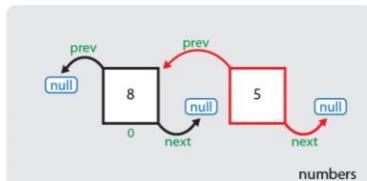


Рис. 4: Добавление второго объекта в связанный список (этап 1)

size	1
first	{null} 8 {null}
last	{8} {null}

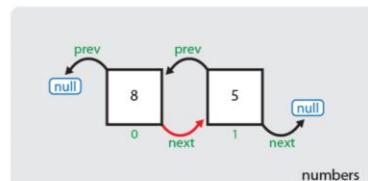


Рис. 5: Добавление второго объекта в связанный список (этап 2)

size	2
first	{null} 8 {5} {null}
last	{8} {5} {null}

3) Добавление объекта в середину связанного списка: `numbers.add(1, 13)`

Как и метод `add(element)`, данный метод вызывает несколько других методов. Сначала осуществляется проверка значения `index`, которое должно быть положительным числом, меньшим или равным размеру списка. Если `index` не удовлетворит этим условиям, то будет сгенерировано исключение `IndexOutOfBoundsException`. Затем, если `index` равен размеру коллекции, то осуществляются действия, описанные в п. 2, так как фактически необходимо вставить элемент в конец существующего списка.

Если же `index` не равен `size` списка, то осуществляется вставка перед элементом, который до этой вставки имеет заданный индекс, т.е. у нас перед узлом со значением 5.

Для начала с помощью метода `node(index)` определяется узел, находящийся в данный момент под индексом, под который нам необходимо вставить новый узел. Поиск этого узла осуществляется с помощью простого цикла **for по половине списка** (в зависимости от значения индекса — либо с начала до элемента, либо с конца до элемента).

Далее создается узел для нового элемента (число 13), ссылка на предыдущий элемент устанавливается на узел, в котором элементом является число 8, а ссылка на следующий элемент устанавливается на узел, в котором элементом является число 5. Ссылки ранее существующих узлов пока не изменены (рис. 6).

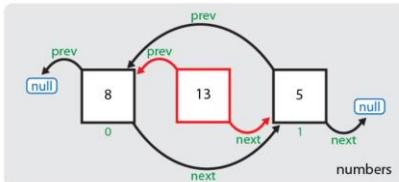


Рис. 6: Добавление объекта в середину связанного списка (этап 1)

size	2
first	{null} 8 {5} {null}
last	{8} {5} {null}

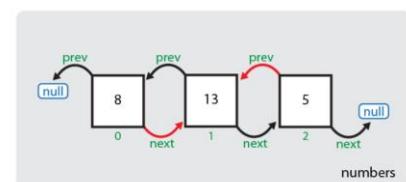


Рис. 7: Добавление объекта в середину связанного списка (этап 2)

size	3
first	{null} 8 {13} {5} {null}
last	{13} {5} {null}

Теперь последовательно заменяются ссылки: для элемента, следующего за новым элементом, заменяется ссылка на предыдущий элемент (теперь она указывает на узел со значением 13), для предшествующего новому элементу заменяется ссылка на следующий элемент (теперь она указывает на узел со значением 5). И в последнюю очередь увеличивается размер списка (рис. 7).

4) Удаление объекта из связанного списка `numbers.remove(Integer.valueOf(5));`

Для удаления одного элемента из списка класс `LinkedList` предлагает нам аж 10 методов, различающихся по типу возвращаемого значения, наличию или отсутствию выбрасываемых исключений, а также способу указания, какой именно элемент следует удалить:

Метод	Удаляемый элемент	Если найден	Если не найден	
poll()	первый	возвращает удаленный элемент	возвращает null	
pollFirst()			NoSuchElementException	
remove()			NoSuchElementException	
removeFirst()			NoSuchElementException	
pollLast()	последний	возвращает null	возвращает null	
removeLast()			NoSuchElementException	
remove(index)	элемент в указанной позиции	возвращает true	IndexOutOfBoundsException	
remove(object)	первое вхождение указанного объекта		возвращает false	
removeFirstOccurrence(object)	последнее вхождение указанного объекта		возвращает false	
removeLastOccurrence(object)				

Табл. 1: Методы *LinkedList* для удаления элемента

Итак, что же происходит при вызове метода **remove(object)**?

Сначала искомый объект сравнивается по порядку со всеми элементами, сохраненными в узлах списка, начиная с нулевого узла. Когда найден узел, элемент которого равен искомому объекту, первым делом элемент сохраняется в отдельной переменной. Потом переопределяются ссылки соседних узлов так, чтобы они указывали друг на друга (рис. 9). Затем обнуляется значение узла, который содержит удаленный объект, а также уменьшается размер коллекции (рис. 10).

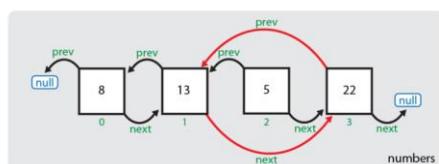


Рис. 9: Удаление элемента из связанных списков по его значению (этап 1)

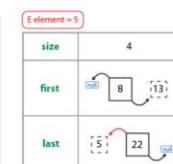
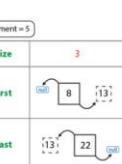


Рис. 10: Удаление элемента из связанных списков по его значению (этап 2)



Когда мы используем метод **remove(index)**, также вызывающий метод **unlink(node)**, то значение данного элемента последовательно возвращается сначала методом **unlink(node)**, а затем и методом **remove(index)**.

Похожая ситуация наблюдается и в остальных методах, возвращающих значение удаленного элемента, только внутри вызываются другие методы, отсоединяющие ссылку: в методах **poll()**, **pollFirst()**, **remove()** и **removeFirst()** это метод **unlinkFirst(node)**, а в методах **pollLast()** и **removeLast()** — метод **unlinkLast(node)**.

Скорость основных операций.

LinkedList знает, где находится его голова и где находится хвост)))

ArrayList

addLast -	O[1]
get -	O[1]
removeFirst -	O[N]
insert -	O[N]
reallocation -	O[N]

LinkedList

addLast -	O[1]
get -	O[N]
removeFirst -	O[1]
insert -	O[1]
reallocation -	O[N]

Remove elements from middle (100k)
LinkedList: 7519 ms
ArrayList: 1544 ms

линикед лист тяжелее (память)
в угоду памяти уменьшить скорость (ArrayList)

	Временная сложность							
	Среднее				Худшее			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)

Performance Test

Operations	ArrayList add	LinkedList add	ArrayList get	LinkedList get	ArrayList remove	LinkedList remove	Вставить	Получить	Удалить
------------	---------------	----------------	---------------	----------------	------------------	-------------------	----------	----------	---------

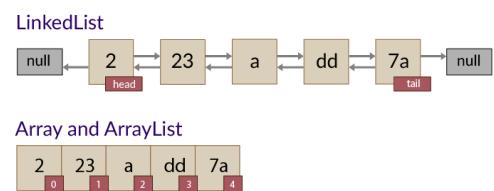
Что следует помнить о **LinkedList**, решая, использовать ли данную коллекцию:

- не синхронизирована
- позволяет хранить любые объекты, в том числе null и повторяющиеся
- за **константное время O(1)** выполняются операции вставки и удаления первого и последнего элемента и операции вставки и удаления элемента из середины списка (не учитывая время поиска позиции элемента, который осуществляется за линейное время)
- за **линейное время O(n)** выполняются операции поиска элемента по индексу и по значению

23. Чем отличаются ArrayList и LinkedList?

Вопрос проверяет знание особенностей реализации ArrayList и LinkedList и эффективности операций в этих разных реализациях.

ArrayList vs. LinkedList

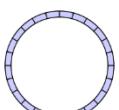


В вопрос иногда добавляют **Vector** – пере-синхронизированный и устаревший вариант **ArrayList**, который лучше заменить **Collections.synchronizedList()**.

ArrayList хранит данные в массиве, **LinkedList** в связанным списке. Из этого вытекает разница в эффективности разных операций:

- **ArrayList** лучше справляется с изменениями в середине и ростом в пределах capacity
- **LinkedList** – на краях. **В целом обычно ArrayList лучше.**

Стоит добавить, что для работы на краях лучше использовать реализации специально для этого спроектированного интерфейса Deque: например, реализующую **кольцевой буфер*** **ArrayDeque**.



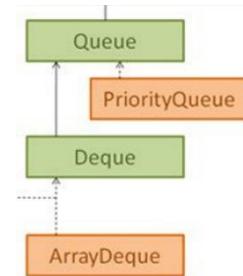
***Кольцевой буфер**, или **циклический буфер** — это структура данных, использующая единственный буфер фиксированного размера таким образом, как будто бы после последнего элемента сразу же снова идет первый. Такая структура легко предоставляет возможность буферизации потоков данных.

24. Что такое Queue?

public interface **Queue<E>** extends [Collection<E>](#)

Queue – это односторонняя очередь, когда элементы можно получить в том порядке, в котором добавляли. FIFO (первым вошёл, первым вышел).

Согласно Javadoc очереди, очередь добавляет следующие методы:



	Throws exception Выдает исключение	Returns special value Возвращает специальное значение
Insert (вставить)	add(e) – добавь	offer(e) - предложить
Remove (удалить)	remove() - удали	poll() – срезать верхушку
Examine	element() – дай элемент	peek() - посмотреть

25. Что такое Deque? Чем отличается от Queue? разница между Queue, Deque и Stack?

Deque (Double Ended Queue) – это двусторонняя очередь, т.е. Можно вставлять/получать элементы как из начала, так и с конца. Расширяет Queue. Согласно документации, это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов (реализация: LIFO, либо FIFO).

Реализации и Deque, и Queue обычно **НЕ переопределяют методы equals() и hashCode()**, вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

java.util
Interface Deque<E>

Type Parameters:

E - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>, Queue<E>

All Known Subinterfaces:

BlockingDeque<E>

All Known Implementing Classes:

ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList

Queue – это односторонняя очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out). Соответственно извлечение элемента осуществляется с начала очереди, а вставка элемента в конец очереди.

Хотя этот принцип нарушает, к примеру **PriorityQueue**, использующая «natural ordering» или переданный Comparator при вставке нового элемента.

Приоритетная очередь

PriorityQueue

- Элементы располагаются в очереди в порядке сравнения: Comparable, natural order, Comparator
- Создайте
 - Queue<Integer> q = LinkedList<>();
 - Заполните элементами от 5 до 1
 - Напечатайте элементы очереди:
 - while (!q.isEmpty()) { q.poll() }
 - Выполните то же самое для PriorityQueue

Приоритетная очередь

Пример

- Создайте очередь чисел, которая сначала возвращает все четные, а потом нечетные числа в порядке возрастания:
 - q.add(5); q.add(2); q.add(1); q.add(4);
 - Poll: 2,4,1,5

26. Приведите пример реализации Deque. https://youtu.be/5_f5foEXiYY

Например, класс ArrayDeque<E>.

Этот класс представляют обобщенную двунаправленную очередь, наследуя функционал от класса `AbstractCollection` и **применяя интерфейс Deque**.

В классе `ArrayDeque` определены следующие конструкторы:

- `ArrayDeque()`: создает пустую очередь
- `ArrayDeque(Collection<? extends E> col)`: создает очередь, наполненную элементами из коллекции `col`
- `ArrayDeque(int capacity)`: создает очередь с начальной емкостью `capacity`.
Если мы явно не указываем начальную емкость, то **емкость по умолчанию будет равна 16**

Пример использования класса:

```
// очередь из объектов Person
ArrayDeque<Person> people = new ArrayDeque<Person>();
people.addFirst(new Person("Tom"));
people.addLast(new Person("Nick"));
// перебор без извлечения
for(Person p : people){

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

27. Какая коллекция реализует FIFO?

FIFO - First-In-First-Out (первый пришел, первым ушел). По этому принципу обычно построена такая структура данных, как очередь (`java.util.Queue`).

28. Какая коллекция реализует LIFO?

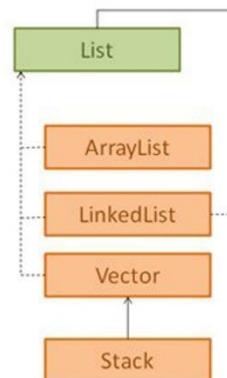
Stack работает по схеме LIFO (последним вошел, первым вышел, как стопка книг). Всякий раз, когда вызывается новый метод, содержащий примитивные значения или ссылки на объекты, то на вершине стека под них выделяется блок памяти.

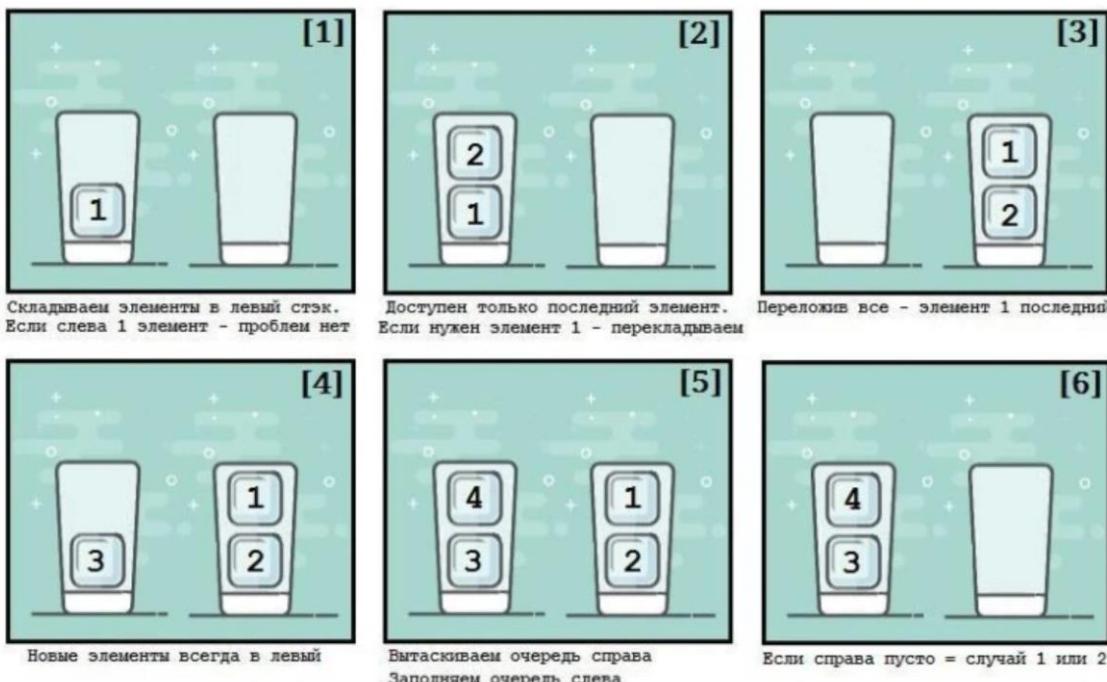
Stack реализует дополнительные методы: `peek` (взглянуть, посмотреть), `pop` (вытолкнуть), `push` (затолкать).

Класс `Vector` является поток безопасным. Это означает, что, если один поток работает над `Vector`, ни один другой поток не сможет его удержать. В отличие от `ArrayList`, только один поток может выполнять операцию по вектору за раз. `ArrayList` не синхронизирован, что означает, что в `ArrayList` одновременно могут работать несколько потоков. Таким образом, вы не получите исключение `ConcurrentModificationException`. **Если потоковая реализация не нужна, рекомендуется использовать `ArrayList` вместо `Vector`.**

Значение Vector по умолчанию удваивает размер его массива, в то время как ArrayList увеличивает его размер массива на 50 процентов. В зависимости от того, как вы используете эти классы, вы можете получить **большой удар производительности при добавлении новых элементов**.

Кстати, на понимание стэков есть интересная задача, описанная в книге "Карьера программиста" (Cracking Coding Interview). Используя структуру "стэк" (LIFO) нужно реализовать структуру "очередь" (FIFO). Вот как это можно сделать:





29. Оцените количество памяти на хранение одного примитива типа byte в LinkedList?

Каждый элемент LinkedList хранит ссылку на предыдущий элемент, следующий элемент и ссылку на данные.

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
    ...
}
```

// 8 байт
// 4 байта
// 4 байта
// 4 байта
Итого 24 байта (кратно 8) для хранения элемента в списке
Плюс 16 байт для хранения упакованного объекта Byte
(8 байт заголовок и 8 байт для значений из пула int).
Итого 40 байт для 32-битной JVM
Итого 64 байта для 64-битной JVM

Для 32-битных систем каждая ссылка занимает 32 бита (4 байта). Сам объект (заголовок) вложенного класса Node занимает 8 байт. $4 \times 3 + 8 = 20$ байт, а т.к. размер каждого объекта в Java кратен 8, соответственно получаем 24 байта. Примитив типа byte занимает 1 байт памяти, но в JCF примитивы упаковываются: объект типа Byte занимает в памяти 16 байт (8 байт на заголовок объекта, 1 байт на поле типа byte и 7 байт для кратности 8). Значения от -128 до 127 кэшируются в пул int и для них новые объекты каждый раз не создаются. Таким образом, в x32 JVM 24 байта тратятся на хранение одного элемента в списке и 16 байт - на хранение упакованного объекта типа Byte. **Итого 40 байт.**

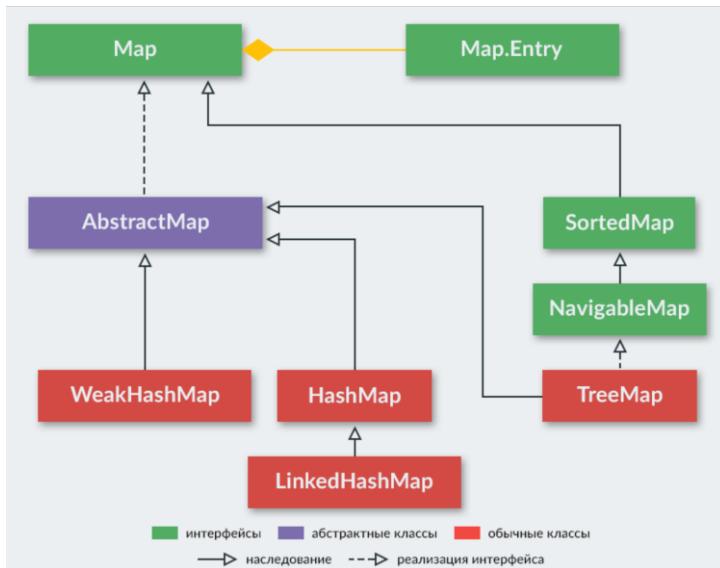
Для 64-битной JVM каждая ссылка занимает 64 бита (8 байт), размер заголовка каждого объекта составляет 16 байт (два машинных слова). Вычисления аналогичны: $8 \times 3 + 16 = 40$ байт и 24 байта. **Итого 64 байта.**

30. Оцените количество памяти на хранение одного примитива типа byte в ArrayList?

ArrayList основан на массиве, для примитивных типов данных осуществляется автоматическая упаковка значения, поэтому 16 байт тратится на хранение упакованного объекта и 4 байта (8 для x64) на хранение ссылки на этот объект в самой структуре данных.

Таким образом, в x32 JVM 4 байта используются на хранение одного элемента и 16 байт на хранение упакованного объекта типа Byte. Для x64 - 8 байт и 24 байта соответственно.

31. Какие существуют реализации Map?



32. Как устроена HashMap? (Расскажите про принцип корзин)

HashMap<K,V> - это **КЛАСС** в Java коллекции интерфейсов и классов (Java Collection Framework), который реализует interface **Map<K,V>** (пары ключей и значений (key & value)):

- КЛЮЧИ элементов должны быть уникальными (и **IMUTABLE**), может быть null.
- ЗНАЧЕНИЯ элементов могут повторяться. Значение может быть null.

HashMap **НЕ запоминает порядок добавления элементов** в коллекцию.
НЕ синхронизируемая! Нельзя использовать в условиях многопоточности, для этого есть **ConcurrentHashMap**.

Почему так популярен? HashMap быстро работает, и большинство операций выполняется за фиксированное или **константное O(1) время** благодаря оптимизированному доступу к данным.

«КАК ЭТО УСТРОЕНО?» — часто спрашивают, посмотреть видео https://youtu.be/kk_9md24Ttk

Словарик HashMap

Хеш-таблица – это структура данных, реализующая интерфейс **ассоциативного массива** (абстрактная модель «ключ – значение» или entry), которая обеспечивает очень быструю вставку и поиск: независимо от количества элементов вставка и поиск (а иногда и удаление) выполняются за время, близкое к константе – **O(1)**.

Хеш-функция hashCode() задаёт связь между значением элемента и его позицией в хеш-таблице.

Table — массив типа **Entry[]** (вход), который является хранилищем для элементов **корзин/бакетов** (хранящих ссылки на списки/цепочки значений).

Size — Количество **нод/узлов** HashMap-а (не путать с ёмкостью массива).

Capacity – это ёмкость созданного массива `Entry[]`, значение по умолчанию **16 элементов** (идентификатор от 0 до 15 мест для корзин или бакетов). Укажем больше, будет больше.

```
// Инициализация хранилища в конструкторе
// capacity - по умолчанию имеет значение 16
table = new Entry[capacity];
```

Корзины или бакеты («buckets») – это элементы массива (ячейки), которые используются для хранения отдельно взятых узлов (или цепочек узлов).

Внутри каждой ячейки массива (корзины или бакета) лежит **односвязный список `LinkedList`**, либо лежит красное-чёрное дерево при перестроении (класс `TreeMap`). Максимальная длина связанного списка – цепочка узлов из 8 значений.

Узел представляет собой объект вложенного (в класс `HashMap`) **класса `Node`** (или `TreeNode` при древовидной структуре), имплементирует интерфейс `Map.Entry<K,V>`, содержит поля: `hash`, `key`, `value`, `next`.

LoadFactor — коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;

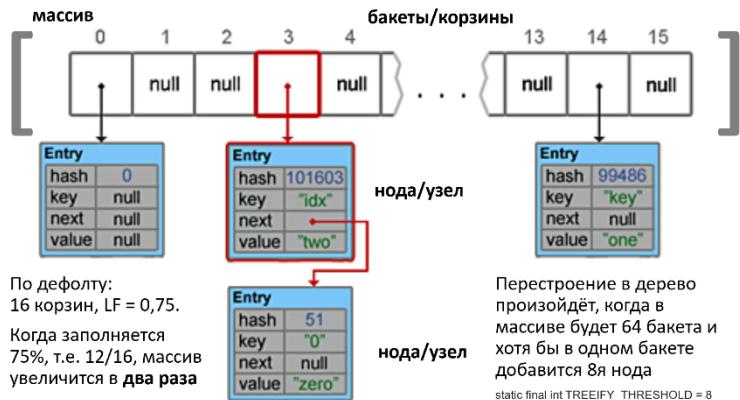
Чем больше LoadFactor, тем больше экономия памяти под массив, но поиск элемента будет занимать также больше времени.

Вы можете указать свои емкость и коэффициент загрузки, используя конструкторы `HashMap(capacity)` и `HashMap(capacity, loadFactor)`. Максимальная емкость, которую вы сможете установить, равна половине максимального значения int. Почему? Тип данных хэш-кода.

Threshold — предельное количество элементов, при достижении которого, размер хеш-таблицы увеличивается вдвое, рассчитывается по формуле (`capacity * loadFactor`).

Структура

В основе `HashMap` лежит массив. Элементами этого массива являются структуры односвязного списка `LinkedList`, которые заполняются элементами.



При помещении нового элемента (`k/v`), ищется нужная корзина. Если в корзине пусто, то помещаем туда новый объект.

Если занято, то находим нужную корзину и проверяем первый объект, который там лежит.

- Если такой элемент в цепочке существует, его значение перезаписывается.
- Если место занято и объект не совпадает с нашим, то размещаем наш объект в следующем узле (она же нода, она же entry) в цепочке списка одной корзины (см. №3), для добавления нового узла/ноды будет вызван метод `addEntry()`.

Нода содержит поля и хранит инфо: `hash`, `key`, `value`, `next` (ссылка на следующий узел).

Хэш-код

Перед тем, как что-то сделать с объектом вычисляется его **НОВЫЙ** хэш-код. Для генерации используется метод `hash(hashCode)`, в который передается `key.hashCode()`.

Формула расчёта ХЭШ-КОДА для ключа:
(hashCode) ^ (hashCode >>> 16) побитовый сдвиг на 16

```
static final int hash(Object key) {  
    int h;  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

Почему бы просто не вычислить код с помощью hashCode()?

Это сделано из-за того, что hashCode() можно реализовать так, что только нижние биты int'a будут заполнены. Например, для Integer, Float – если мы в HashMap кладем маленькие значения, то у них и биты хеш-кодов будут заполнены только нижние. В таком случае ключи в HashMap будут иметь тенденцию скапливаться в нижних ячейках, а верхние будут оставаться пустыми, что не очень эффективно.

На то, в какой бакет попадёт новая запись, влияют только младшие биты хеша. Поэтому и придумали различными манипуляциями подмешивать старшие биты хеша в младшие, чтобы улучшить распределение по бакетам (чтобы старшие биты родного хеша объекта начали вносить корректиды в то, в какой бакет попадёт объект) и, как следствие, производительность. Потому и придумана дополнительная функция hash внутри HashMap.

Пример (h – hashCode):

В моем случае, для ключа со значением "0" метод hashCode() вернул значение 48, в итоге:

```
h ^ (h >>> 20) ^ (h >>> 12) = 48  
h ^ (h >>> 7) ^ (h >>> 4) = 51
```

При значении хэша **51** и размере таблицы **16**, мы получаем индекс корзины в массиве:

$h \& (length - 1) = 3$

3. С помощью метода indexFor(hash, tableLength), определяется позиция в массиве, куда будет помещен элемент.

```
static int indexFor(int h, int length)  
{  
    return h & (length - 1);  
}
```

Эта формула применяется и для определения бакета при добавлении элемента и для поиска бакета, когда пытаемся достать элемент (пару k/v).

Вопрос: как узнать, в какую корзину попадёт элемент, если значение его хэш-кода 51?

Ответ: остаток от деления на кол-во корзин (т.к. $51 \% 16 = 3$ (проверка $3 * 16 = 48 + 3$))

[Как происходит сравнение при добавлении элемента? Аналогично при поиске \(получить\).](#)

Рассчитав и зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой корзине (ячейке). Хэш и ключ нового элемента поочередно сравниваются с хэшами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается.

Сравниваем КЛЮЧ нашего объекта с первым в цепочке объектом:

- проверка по хэш-коду
- ссылочное сравнение
- если предыдущий пункт выдал false, то проверка на equals

Иногда задают вопрос, где используется equals при добавлении объекта в мапу. Вот тут как раз)))
Т.к. equals - дорогая операция, поэтому в самом конце.

```
if (e.hash == hash && (e.key == key || key.equals(e.key)))  
{  
    V oldValue = e.value;  
    e.value = value;  
  
    return oldValue;  
}
```

Если же предыдущий шаг не выявил совпадений, будет вызван метод **addEntry(hash, key, value, index)** для добавления нового элемента.

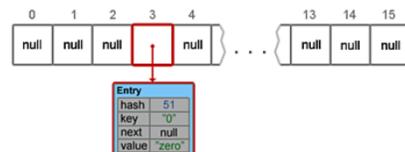
Если при добавлении элемента в качестве ключа был передан null, действия будут отличаться. Будет вызван метод **putForNullKey(value)**, внутри которого нет вызова методов **hash()** и **indexFor()** (потому как все элементы с null-ключами всегда помещаются в корзину[0]), но есть такие действия:

Все элементы цепочки, привязанные к корзине[0], поочередно просматриваются в поисках элемента с ключом null.

Если такой элемент в цепочке существует, его значение перезаписывается.

Если элемент с ключом null не был найден, будет вызван уже знакомый метод **addEntry()**.

```
void addEntry(int hash, K key, V value, int index)
{
    Entry<K, V> e = table[index];
    table[index] = new Entry<K, V>(hash, key, value, e);
    ...
}
```



Если при добавлении элемента возникает коллизия.

Ситуация, когда разные ключи попадают в один и тот же бакет (даже с разными хешами), называется **коллизией или столкновением**. Даже если хеш-таблица больше, чем набор данных, и была выбрана хорошая хеш-функция, это не гарантирует того, что коллизии не возникнут. Да и значение хеша ограничено диапазоном значений типа **int** (порядка 4 млрд.). Полученное новое значение также нужно куда-то записать, и для этого нужно определить, куда именно оно будет записано. Это называется решением коллизии. Существует два подхода:

external chaining или метод цепочек (реализован в **HashMap**) — т.е. в ячейке на самом деле содержится список (chain). А уже в списке может содержаться несколько значений (не обязательно с одинаковым хеш-кодом).

linear probing или метод открытой адресации (реализован в **IdentityHashMap**) — заключается в поиске первой пустой ячейки после той, на которую указала хеш-функция;

1) Что такое бинарное дерево?

Двоичное/бинарное дерево, что это?

Ищем 16.

16 > 10? значит идём вправо

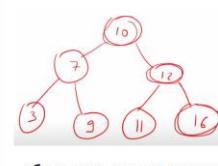
16 > 12? значит идём вправо

нашли!

БИНАРНЫЙ ПОИСК (два значения сравниваем)
Скорость будет $O(\log n)$

У двоичных деревьев может отсутствовать баланс.

Сбалансированное дерево (справа всегда больший элемент, слева меньший)



Дерево переставит свои элементы (с помощью определённого алгоритма), оптимизируется, СБАЛАНСИРУЕТСЯ, чтобы облегчить поиск.

И получится СБАЛАНСИРОВАННОЕ ДВОИЧНОЕ ДЕРЕВО
Зачем? позволяет быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла.

Двоичное дерево — структура данных, в которой **каждый узел** (родительский) **имеет не более двух потомков** (правый и левый наследник).

Двоичное дерево поиска строится по определенным правилам:

- каждый узел имеет не более двух детей;
- каждое значение, **меньшее**, чем значение узла, становится левым ребенком или ребенком левого ребенка;

2) Когда и как происходит перестроение карты в дерево?

Первоначальный размер **HashMap** = 16 бакетов (напомню, бакет — это ячейка массива).

Когда массив заполняется на 75%, то есть заполняются 12/16 бакетов ($16 \times 0,75 = 12$, т.к. **loadFactor = 0,75**), размер массива **увеличивается в 2 раза**, т.е. становится 32 бакета.

И так далее (64, 128 ...). При увеличении размера массива все объекты, уже содержащиеся в `HashMap`, будут перераспределены по новым бакетам, с учётом их нового количества.

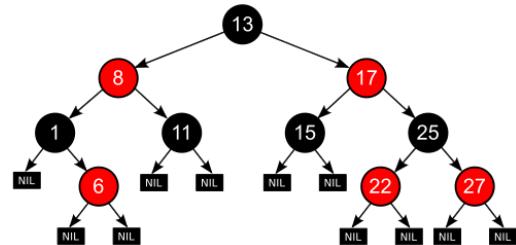
Каждый бакет содержит в себе ноды (пары ключ-значение), когда нодов становится 8, а бакетов 64, то структура `Node` перестраивается в **красно-чёрное дерево** (`TreeNode`).

Обратное перестроение `TreeNode` → `Node` случается, если количество нод в цепочке < 6

Протестировать визуализацию дерева <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

Каковы особенности красно-чёрного дерева?

- 1) Оба поддерева являются **бинарными деревьями поиска**
- 2) Левые потомки должны быть меньше своего корневого узла (или равны ему), а правый узел всегда больше левого (за счёт этого обеспечивается быстрый бинарный поиск)
- 3) Каждый узел окрашен либо в красный, либо в чёрный цвет (в структуре данных узла появляется доп. поле – бит цвета).
- 4) Корень и листья (так называемые `NIL`-узлы) окрашены в чёрный цвет.
- 5) Каждый красный узел должен иметь два чёрных дочерних узла. Красные узлы в качестве дочерних могут иметь только чёрные.
- 6) Пути от узла к его листьям должны содержать **одинаковое количество чёрных узлов (это черная высота)**.
- 7) При нарушении этого порядка дерево пере-балансируется.



ПРОИЗВОДИТЕЛЬНОСТЬ `HashMap`

Чем лучше реализован `HashCode`, тем лучше будут использоваться бакеты/корзинки/ячейки памяти.

ПРАВИЛО: для того, чтобы вставить элемент или получить его, требуется **константное время $O(1)$ SUPER**

При перестроении в красно-чёрное дерево, т.к. древовидная структура, скорость будет $O(\log n)$ - **GOOD**

НО! Иногда будет требоваться больше времени, т.к. внутри `HashMap` связанный `LinkedList`, **линейное $O(n)$ OK**

Метод `get()` и `put()` зависят напрямую от реализации `HashCode`

элемент 1 будет найден сразу, а элемент №100 нет и времени потребуется больше, т.к. список связанный

N.B. Big-O для реализаций Java Collections

Т.е.: «одна операция для всех возможных входных данных» — $O(1)$.

$O(1)$ можно прочитать как «сложность порядка 1» (*order 1*),

или «алгоритм выполняется за постоянное/константное время» (*constant time*).

$O(1)$ алгоритмы самые эффективные, например

`const nums = [1,2,3,4,5];`

`const firstNumber = nums[0];`

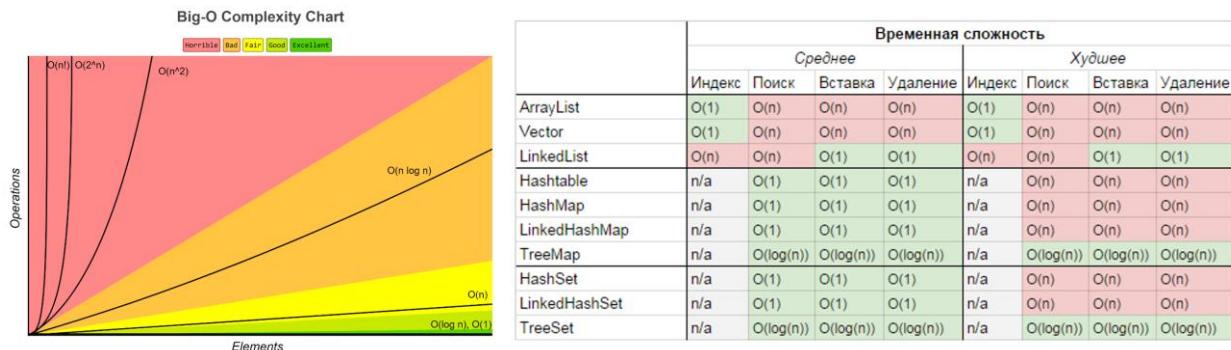
Красная зона – ужасно $O(n^2)$; $O(2^n)$; $O(n!)$
Оранжевая – плохо: $O(n \log n)$

Жёлтая – справедливо, оправдано: **$O(n)$ или линейное время**

Салатовый – хорошо **$O(\log n)$**

Зелёный – превосходно! **$O(1)$ или константа**

Сравнение производительности разных коллекций по ссылке (аккуратно, можно зависнуть): <https://www.bigocheatsheet.com/>



32.1. Сложность основных операций HashMap?

Что имеется в виду? <https://javarush.ru/groups/posts/2496-podrobnihy-razbor-klassa-hashmap>

Что происходит внутри **HashMap.put()**?

1. **Вычисляется хэш ключа.** Если ключ `null`, хэш считается равным `0`. Чтобы достичь лучшего распределения, результат вызова `hashCode()` «перемешивается»: его старшие биты XOR-яются на младшие. `XOR (^)` – это математический оператор. Применяется к логическим операциям.
2. Значения внутри хэш-таблицы хранятся в специальных структурах данных – нодах, в массиве. **Из хэша высчитывается номер бакета** – индекс для значения в этом массиве. Полученный хэш обрезается по текущей длине массива. Длина – всегда степень двойки, так что для скорости используется битовая операция `&`.
3. **В бакете ищется нода.** В ячейке массива лежит не просто одна нода, а связка всех нод, которые туда попали. Исполнение проходит по этой связке (цепочке или дереву), и ищет ноду с таким же ключом. Ключ сравнивается с имеющимися сначала на `==`, затем на `equals`.
4. **Если нода найдена – её значение просто заменяется новым.**
Работа метода на этом завершается.
5. **Если ноды с таким же ключом в бакете пока нет** – добавляемая пара ключ-значение **запаковывается в новый объект типа Node**, и прикрепляется к структуре существующих нод бакета. Ноды составляют структуру за счет того, что в ноде хранится ссылка на следующий элемент (для дерева – следующие элементы). Кроме самой пары и ссылок, чтобы потом не считать заново, **записывается и хэш ключа**.
7. В случае, когда структурой была цепочка а не дерево, и длина цепочки превысила 7 элементов – происходит процедура **treeification** – превращение списка в самобалансирующееся дерево. В случае коллизии это ускоряет доступ к элементам на чтение с $O(n)$ до $O(\log(n))$. У comparable-ключей для балансировки используется их естественный порядок. Другие ключи балансируются по порядку имен их классов и значениям `identityHashCode`-ов. Для маленьких хэш-таблиц (< 64 бакетов) «одеревенение» заменяется увеличением (см. п.8).
8. Если новая нода попала в пустую ячейку, заняла новый бакет – **увеличивается счетчик структурных модификаций**. Изменение этого счетчика сообщит всем итераторам контейнера, что при следующем обращении они должны выбросить `ConcurrentModificationException`.
9. Когда количество занятых бакетов массива превысило пороговое (`capacity * load factor`), внутренний **массив увеличивается вдвое**, а для всего содержимого выполняется **рехэш** – все имеющиеся ноды перераспределяются по бакетам по тем же правилам, но уже с учетом нового размера.

Закрепление или самопроверка <https://itsobes.ru/JavaSobes/tags/kollektsii/>

Один из популярнейших вопросов, потому что содержит много нюансов. Лучше всего подготовиться к нему помогает чтение исходного кода `HashMap`. Реализация подробно рассмотрена во множестве статей, например [на хабре](#).

Нюансы которые стоит повторить и запомнить:

- Общий принцип: внутренний массив `table`, содержащий бакеты (корзины) – списки элементов с одинаковыми пересчитанными хэш-суммами;
- Пересчет хэш-суммы для умещения `int` индексов в `capacity` ячейках `table`;
- `rehash` – удвоение размера `table` при достижении `threshold` (`capacity*loadFactor`) занятых бакетов;
- Невозможность сжать однажды раздувшийся `table`;
- [Два способа](#) разрешения коллизий: используемый в `HashMap` метод цепочек и альтернатива – открытая адресация;
- Варианты для многопоточного использования: пересинхронизированная `Hashtable` и умная `ConcurrentHashMap`;
- Оптимизация Java 8: превращение списка в бакете в дерево при достижении 8 элементов – при большом количестве коллизий скорость доступа растет с $O(n)$ до $O(\log(n))$;
- Явное использование бакета 0 для ключа `null`;
- Связь с `HashSet` – `HashMap`, в котором используются только ключи;
- Нет гарантий порядка элементов;

Обсуждая этот вопрос на интервью вы обязательно затронете особенности методов `equals/hashCode`. Возможно придется поговорить об альтернативных хранилищах ключ-значение – `TreeMap`, `LinkedHashMap`.

33. Что такое `LinkedHashMap`?

`LinkedHashMap` - что в нем от `LinkedList`, а что от `HashMap`?

Реализация `LinkedHashMap` отличается от `HashMap` **поддержкой двухсвязного списка**, определяющего порядок итерации по элементам структуры данных.

По умолчанию элементы списка упорядочены согласно их порядку добавления в `LinkedHashMap` (insertion-order). Однако порядок итерации можно изменить, установив параметр конструктора `accessOrder` в значение `true`. В этом случае доступ осуществляется по порядку последнего обращения к элементу (access-order).

Это означает, что при вызове методов `get()` или `put()` элемент, к которому обращаемся, перемещается в конец списка.

При добавлении элемента, который уже присутствует в `LinkedHashMap` (т.е. с одинаковым ключом), порядок итерации по элементам не изменяется.

Задачки, что будет в консоли (ответ в жёлтой заливке)? <https://youtu.be/5lu4ZUcrJ0g>

Порядок в `HashMap` не гарантируется, в отличие от `LinkedHashMap`). Поля и конструктор (`capacity`, `loadFactor`, `order`). Класс несинхронизированный (многопоточность не поддерживает).

Порядок элементов в Map

`LinkedHashMap`

- Создайте `HashMap<Integer, String> m`
- Заполните ключами от 5 до 1
 - `m.put(5, "a"); m.put(4, "b"); m.put(3, "c"); ...`
- Выполните `System.out.println(map);`

1=e, 2=d, 3=c, 4=b, 5=a
- Проделайте то же самое для `LinkedHashMap<Integer, String> map`

5=a, 4=b, 3=c, 2=d, 1=e

Порядок по доступу

`Access order`

```
public LinkedHashMap(  
    int initialCapacity, // начальный размер  
    float loadFactor, // фактор загрузки  
    boolean accessOrder // порядок по доступу  
)  
  
● false - (по умолчанию) порядок согласно вставке в Map  
● true - порядок согласно частоте доступа (от меньшей к большей)
```

Порядок по доступу

Access order

- Влияют **только** методы `get` и `put`
 - В предыдущем примере замените тар на `LinkedHashMap(5, 1, true)`
 - Перед выводом (`System.out`) выполните доступ к элементам: `3, 5, 1 (map.get(...))`
 - Можно использовать как простейший LRU-контейнер (например, LRU-Cache)
- ```
protected boolean removeEldestEntry(Entry<K, V> eldest) { return this.size() > capacity; }
```

## Простой LRU кеш

Simple LRU-Cache

```
public class SimpleLRUCache<K, V> extends LinkedHashMap<K, V> {
 private final int capacity;

 public SimpleLRUCache(int capacity) {
 super(capacity + 1, 1.0f, true);
 this.capacity = capacity;
 }

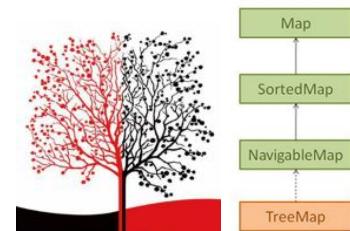
 @Override
 protected boolean removeEldestEntry(Entry<K, V> eldest) {
 return this.size() > capacity;
 }
}
```

SimpleLRUCache(2);  
**put: 1, 2, 3**  
**get: 2**  
**put: 9**  
**{ 2, 9 }**  
print(map);

## 34. Как устроена TreeMap, сложность основных операций?

Реализация интерфейса Map — TreeMap позволяет хранить данные в структурированном виде с возможностью навигации.

Древовидная структура: под капотом TreeMap использует структуру данных, которая называется **красно-чёрное дерево**.



Имплементируя интерфейсы NavigableMap и SortedMap, TreeMap получает дополнительный функционал, которого нет в HashMap, но **плата за это — производительность**.

Хоть класс TreeMap является **самым многофункциональным**, он не всегда может хранить null в качестве ключа. Кроме этого, время доступа к элементам TreeMap будет **самым длительным**. Поэтому если **НЕ нужно хранить данные в отсортированном виде**, лучше использовать HashMap или LinkedHashMap.

|                                          | HashMap                                                               | LinkedHashMap        | TreeMap                                                   |
|------------------------------------------|-----------------------------------------------------------------------|----------------------|-----------------------------------------------------------|
| Порядок хранения данных                  | Случайный. Нет гарантий, что порядок сохранится на протяжении времени | В порядке добавления | В порядке возрастания или исходя из заданного компаратора |
| Время доступа к элементам                | O(1)                                                                  | O(1)                 | O(log(n))                                                 |
| Имплементированные интерфейсы            | Map                                                                   | Map                  | NavigableMap<br>SortedMap<br>Map                          |
| Имплементация на основе структуры данных | Корзины (buckets)                                                     | Корзины (buckets)    | Красно-чёрное дерево (Red-Black Tree)                     |
| Возможность работы с null-ключом         | Можно                                                                 | Можно                | Можно, если используется компаратор, разрешающий null     |
| Потокобезопасна                          | Нет                                                                   | Нет                  | Нет                                                       |

Пример использования TreeMap: <https://javarush.ru/groups/posts/2584-osobennosti-treemap>

Маркетинговый отдел большой компании.

Есть база людей, которым нужно показывать рекламу.

При этом есть два нюанса:

- 1) необходимо вести учет количества показов каждому человеку
- 2) алгоритм показа рекламы для несовершеннолетних отличается

Создадим класс Person, в котором будет храниться вся доступная нам информация о человеке.

В классе Main реализуем логику: создаем TreeMap, где key — это конкретный человек, а value — количество показов рекламы в этом месяце.

В конструкторе передаем компаратор, который отсортирует людей по возрасту. Заполняем map случайными значениями. Теперь нужно получить ссылку на первого взрослого человека в нашем мини-хранилище данных. Делаем это с помощью Stream API. После этого **получаем две независимые мапы, которые передаем в методы показа рекламы.**

## 35. Что такое WeakHashMap?

### Другие реализации Map

- **EnumMap**

- Ключи - это элементы Enum
- Значения хранятся в массиве
- Компактно и эффективно

- **WeakHashMap**

- Слабые ключи
- Элемент удаляется, когда ключ больше не используется (garbage collected)
- Реализация временного хранилища (кеша) ассоциированных данных

### Пример WeakHashMap

```
Map<Data, String> map = new WeakHashMap<Data, String>();
Data data = new Data(); // какой-то объект
map.put(data, "information");

data = null; // делаем доступным для gc()
System.gc();

for (int i = 1; i < 10000; i++) {
 if (map.isEmpty()) {
 System.out.println("Empty!");
 break;
 }
}
Вместо класса Data
можно просто сделать
Object data = new Object();
```

**В чем разница между HashMap и WeakHashMap? Для чего используется WeakHashMap?**

В Java существует 4 типа ссылок:

- сильные (Strong reference)
- мягкие (SoftReference)
- слабые (WeakReference)
- фантомные (PhantomReference)

Особенности каждого типа ссылок связаны с работой Garbage Collector. Если объект можно достать только с помощью цепочки WeakReference (то есть на него отсутствуют сильные и мягкие ссылки), то данный объект будет помечен на удаление.

**WeakHashMap** — это структура данных, реализующая интерфейс Map и основанная на использовании WeakReference для хранения ключей.

Таким образом, пара «ключ-значение» будет удалена из WeakHashMap, если на объект-ключ более не имеется сильных ссылок.

В качестве примера использования такой структуры данных можно привести следующую ситуацию: допустим имеются объекты, которые необходимо расширить дополнительной информацией, при этом изменение класса этих объектов нежелательно, либо невозможно. Добавляем каждый объект в WeakHashMap в качестве ключа, а в качестве значения — нужную информацию. Пока на объект имеется сильная ссылка (либо мягкая), можно проверять хеш-таблицу и извлекать информацию. Как только объект будет удален, то WeakReference для этого ключа будет помещен в ReferenceQueue и затем соответствующая запись для этой слабой ссылки будет удалена из WeakHashMap.

В `WeakHashMap` используются `WeakReferences`, а почему бы не создать `SoftHashMap` на `SoftReferences`?

`SoftHashMap` представлена в сторонних библиотеках, например, в `Apache Commons`.

В `WeakHashMap` используются `WeakReferences`, а почему бы не создать `PhantomHashMap` на `PhantomReferences`?

**PhantomReference** при вызове метода `get()` возвращает всегда `null` (объект недостижим), поэтому тяжело представить назначение такой структуры данных. Фантомные ссылки — это безопасный способ узнать, что объект удален из памяти.

36. Как работает `HashMap` при попытке сохранить в него два элемента по ключам с одинаковым `hashCode()`, но для которых `equals() == false`?

По значению `hashCode()` вычисляется индекс ячейки массива, в список которой этот элемент будет добавлен. Перед добавлением осуществляется проверка на наличие элементов в этой ячейке. Если элементы с таким `hashCode()` уже присутствует, но их `equals()` методы не равны, то элемент будет добавлен в конец списка.

37. Что будет, если мы кладем в `HashMap` ключ, у которого `equals` и `hashCode` определены некорректно?

**Почему бы просто не вычислить код с помощью `hashCode()`?**

Это сделано из-за того, что `hashCode()` можно реализовать так, что только нижние биты `int'a` будут заполнены. Например, для `Integer`, `Float` – если мы в `HashMap` кладем маленькие значения, то у них и биты хеш-кодов будут заполнены только нижние. В таком случае ключи в `HashMap` будут иметь тенденцию скапливаться в нижних ячейках, а верхние будут оставаться пустыми, что не очень эффективно.

На то, в какой бакет попадёт новая запись, влияют только младшие биты хеша. Поэтому и придумали различными манипуляциями подмешивать старшие биты хеша в младшие, чтобы улучшить распределение по бакетам (чтобы старшие биты родного хеша объекта начали вносить корректиды в то, в какой бакет попадёт объект) и, как следствие, производительность. Потому и придумана дополнительная функция `hash` внутри `HashMap`.

38. Возможна ли ситуация, когда `HashMap` выродится в список даже с ключами имеющими разные `hashCode()`?

Это возможно в случае, если метод, определяющий номер корзины будет возвращать одинаковые значения.

39. Почему нельзя использовать `byte[]` в качестве ключа в `HashMap`?

Хэш-код массива не зависит от хранимых в нем элементов, а **присваивается при создании массива** (метод вычисления хэш-кода массива не переопределен и вычисляется по стандартному `Object.hashCode()` на основании адреса массива).

Также у массивов не переопределен `equals` и выполняется сравнение указателей. Это приводит к тому, что обратиться к сохраненному с ключом-массивом элементу не получится при использовании другого массива такого же размера и с такими же элементами, доступ можно осуществить лишь в одном случае — **при использовании той же самой ссылки на массив**, что использовалась для сохранения элемента.

Потому что `new byte[] {1} != new byte[] {1}`. Так же, их метод для получения хеш кода унаследованы от `Object`'а, что значит, что **каждый новый массив имеет новый хеш код**. Следовательно, когда, мы спросим у хеш карты значение по ключу `new byte[] {1}`, то получим `null`, даже если там есть значение по ключу массива байтов с единичной.

## 40. Будет ли работать HashMap, если все добавляемые ключи будут иметь одинаковый hashCode()?

Да, будет, но в этом случае HashMap вырождается в связный список и теряет свои преимущества.

41. Какое худшее время работы метода get(key) для ключа, которого НЕТ в HashMap?
42. Какое худшее время работы метода get(key) для ключа, который ЕСТЬ в HashMap?

O(N). Худший случай – это поиск ключа в HashMap, вырожденного в список по причине совпадения ключей по hashCode(). Для выяснения хранится ли элемент с определенным ключом может потребоваться перебор всего списка.

### Функциональные интерфейсы

1. Что такое функциональный интерфейс?

Интерфейс называется функциональным, если в нём **ровно один абстрактный метод** (то есть описание метода без тела).

```
package java.util;

@FunctionalInterface
public interface Comparator<T> {
 int compare(T o1, T o2);
 // any number of default or static methods
}
```

Ограничений по количеству методов и полей в функциональном интерфейсе нет (кроме абстрактного метода). Многие функциональные интерфейсы, предоставляемые Java8, находятся в пакете **java.util.function**

Может ли функциональный интерфейс содержать что-то кроме абстрактного метода?

Что такое default методы?

Функциональный интерфейс может содержать любое количество методов по умолчанию (default) или статических методов.

default – реализованные методы в функциональных интерфейсах.

При наследовании интерфейса можно переопределить эти методы или же оставить всё как есть (оставить логику по умолчанию).

Статические методы в интерфейсе помогают обеспечивать безопасность, не позволяя классам, которые реализуют интерфейс, переопределить их.

### Зачем?

Когда в Java нужно передать в метод кусочек программной логики, то метод объявляется, как принимающий экземпляр какого-то класса или интерфейса.

2. Для чего нужна аннотация @FunctionalInterface?

Интерфейсы в этом пакете снабжены аннотациями @FunctionalInterface

Аннотация помогает уловить замысел проекта и заручиться **помощью компилятора** в выявлении случайных нарушений замысла проекта.

Если же в интерфейсе с данной аннотацией более одного не реализованного (абстрактного) метода, компилятор не пропустит данный интерфейс, так как будет воспринимать его как ошибочный код.

Интерфейсы и без данной аннотации могут считаться функциональными и будут работать, а @FunctionalInterface является не более чем дополнительной страховкой.

3. Какие встроенные функциональные интерфейсы вы знаете?

Predicate<T> принимает значение, отдаёт булевское true или false

Функциональный интерфейс `Predicate<T>` проверяет **соблюдение некоторого условия**. Если оно соблюдается, то возвращается значение `true`.

В качестве параметра лямбда-выражение принимает объект типа `T`:

```
Predicate<Student> p1 = student -> student.avgGrade > 7.5;
Predicate<Student> p2 = student -> student.sex == 'm';
```

фильтр по двум параметрам (средний балл и пол)

```
package java.util.function;
@FunctionalInterface
public interface Predicate<T> {
 boolean test(T t);
}
```

IntP  
LongP  
DoubleP

BiPredicate<sup>+</sup>

**Операторы – это частный случай функции, когда на входе и на выходе значения одного и того же типа**

### UnaryOperator<T> унарный оператор

`UnaryOperator<T>` принимает в качестве параметра **ОДИН объект** типа `T`, выполняет над ними операции и возвращает результат операций в виде объекта типа `T`.

```
package java.util.function;
@FunctionalInterface
public interface UnaryOperator<T>
 extends Function<T, T> {
 // apply is inherited from Function
}
```

BinaryOperator:  
Int Long Double

### BinaryOperator<T> бинарный оператор

`BinaryOperator<T>` принимает в качестве параметра **ДВА объекта** типа `T`, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа `T`.

### Function<T,R>

**функции принимают/возвращают**

Т - входной параметр (принимает);  
R - return type (возвращает)

```
package java.util.function;
@FunctionalInterface
public interface Function<T, R> {
 BiFunction<T, T, R> apply(T t);
}
```

DoubleF: double → T  
LongToIntF: long → int  
ToIntFunction: T → int

Функциональный интерфейс `Function<T,R>` представляет функцию перехода от объекта типа `T` к объекту типа `R`.

```
Function<Student, Double> f = student -> student.avgGrade;
```

функциональный интерфейс Function

```
Function<Student, Double> f = student -> student.avgGrade;
double res = avgOfSmth(students, stud -> (double)stud.age);
System.out.println(res);
```

принимает → отдаёт

#### Отличие `BinaryOperator` от `Function`

`Function` (функция) используется для преобразования входного параметра или в двух параметров (для `BiFunction`) в какое-либо значение, тип значение может не совпадать с типом входных параметров.

`BinaryOperator` это разновидность `Function`, в которых входные и выходные обобщенные параметры должны совпадать.

Если заглянуть в пакет `java.util.function`, то можно заметить, что `UnaryOperator` расширяет `Function`, а `BinaryOperator` расширяет `BiFunction`.

## Consumer<T> потребитель только принимает

Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая:

```
import java.util.function.Supplier;
import java.util.function.Consumer; // потребитель берёт, но ничего не ВОЗВРАЩАЕТ

public class Test3 {
 public static ArrayList<Car> createThreeCars(Supplier<Car> carSupplier) {
 ArrayList<Car> al = new ArrayList<>();
 for (int i = 0; i < 3; i++) {
 al.add(carSupplier.get());
 }
 return al;
 }

 public static void changeCar(Car car, Consumer<Car> carConsumer) {
 carConsumer.accept(car);
 }
}
```

changeCar(ourCars.get(i), car -> {  
 car.color = "red";  
 car.engine = 2.4;  
 System.out.println("upgraded car: " + car);  
});  
System.out.println("Our cars: " + ourCars);

```
package java.util.function;
@FunctionalInterface
public interface Consumer<T> {
 void accept(T t);
}

IntC
LongC
DoubleC
```

BiConsumer<T,U>:  
void + accept(T,U)

## Supplier<T> поставщик только поставляет

Supplier<T> не принимает никаких аргументов, но должен возвращать объект типа T:

```
import java.util.ArrayList;
import java.util.function.Supplier;

public class Test3 {
 public static ArrayList<Car> createThreeCars(Supplier<Car> carSupplier) {
 ArrayList<Car> al = new ArrayList<>();
 for (int i = 0; i < 3; i++) {
 al.add(carSupplier.get());
 }
 return al;
 }

 public static void main(String[] args) {
 ArrayList<Car> ourCars = createThreeCars(() ->
 new Car(model: "Nissan Tiida", color: "Silver Metalic", engine: 1.6));
 System.out.println("Our cars: " + ourCars);
 }
}
```

Поставщик Supplier (функция интерфейс)  
поставляет машины

```
package java.util.function;
@FunctionalInterface
public interface Supplier<T> {
 T get();
}
```

BooleanS  
IntS  
LongS  
doubleS

## Все способы реализации функционального интерфейса?

В Java 8 функциональные интерфейсы могут быть представлены с использованием лямбда-выражений, ссылок на методы и ссылок на конструкторы.

```
public classForEach {
 public static void main(String[] args) {
 List<String> list = List.of("privet", "kak dela?", "normalno", "poka");
 // ...
 list.forEach(str -> System.out.println(str));
 }
}
```

сообщаем "Потребителю", как нужно использовать элемент коллекции (печатай)

Output:  
privet  
kak dela?  
normalno  
poka

```
list2.add(1);
Iterable
public void forEach(Consumer<? super Integer> action)
{
 list2.forEach(el ->{
 System.out.println(el);
 el*=2;
 System.out.println(el);
 });
}
```

метод принимает параметры интерфейса Consumer

## 4. Что такое ссылка на метод?

Если лямбда выражения вызывают только один существующий метод, лучше ссылать на этот метод по его имени. Ссылки на методы (Method References) – это **компактные лямбда выражения для методов, у которых уже есть имя**.

Например:

```
Consumer consumer = str -> System.out.println(str);
можно переписать с помощью method references:
Consumer consumer = System.out :: println;
```

## 5. Что такое лямбда-выражение? Чем его можно заменить?

Лямбда-выражение или просто лямбда в Java — **упрощённая запись анонимного класса**, реализующего ТОЛЬКО функциональный интерфейс.



Лямбда представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызывать в различных местах программы. Основу лямбда-выражения составляет **лямбда-оператор**, который представляет стрелку `->`

Синтаксис лямбда выражения имеет вид: **параметры `->` {тело функции}; // заменяет анонимный класс (нет названия класса, но есть параметры и тело метода)**

// внутри параметр-листа мы записываем метод

Лямбда-выражение (слева) представляет сокращенную запись анонимного класса (справа)

|                                                                                                                                                                                                  |                                                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ShapeService rectangleService = (double a, double b) -&gt; {<br/>    return 2 * (a + b);<br/>};<br/>или<br/>ShapeService rectangleService = (double a, double b) -&gt; 2 * (a + b);</code> | <code>ShapeService rectangleService = new ShapeService() {<br/>    @Override<br/>    public double perimeter(double a, double b) {<br/>        return 2 * (a + b);<br/>    }<br/>};</code> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Лямбда vs анонимные классы

Лямбда выражения являются альтернативой анонимным классам. Но они не одинаковы.

#### Общее:

- Локальные переменные могут быть использованы только если они **final** или **effective final**.
- Разрешается доступ к переменным класса и статическим переменным класса.
- Они не должны выбрасывать больше исключений чем определено в **throws** метода функционального интерфейса.

#### Различия:

- Для анонимных классов ключевое слово **this** ссылается на сам класс. Для лямбды выражений на внешний класс.
  - Анонимные классы компилируются во внутренние классы.
- Но лямбда выражения преобразуются в статические **private** методы класса, в котором они используют **invokedynamic** инструкцию.
- Лямбда более эффективны, так как не надо загружать еще один класс.

Самый короткий вариант написания лямбда выражения:  
`stud -> stud.avgGrade > 8.5`

Вы можете использовать смешанный вариант написания лямбда выражения:  
слева от оператора стрелка писать короткий вариант, справа — полный. Или наоборот.

Более полный вариант написания лямбда выражения:  
`(Student stud) -> {return stud.avgGrade > 8.5;}`

Если вы используете полный вариант написания для части лямбда выражения справа от стрелки, то вы должны использовать слово **return** и знак `«;»`

В лямбда выражении оператор стрелка разделяет параметры метода и тело метода.

Левая часть лямбда выражения может быть написана в краткой форме, если метод интерфейса принимает только 1 параметр. Даже если метод интерфейса принимает 1 параметр, но в лямбда выражении вы хотите писать данный параметр используя его тип данных, тогда уже вы должны писать левую часть лямбда выражения в скобках.

В лямбда выражении справа от оператора стрелка находится тело метода, которое было бы у метода соответствующего класса, имплементировавшего наш интерфейс с единственным методом.

Если в правой части лямбда выражения вы пишите более одного statement-a, то вы должны использовать его полный вариант написания.

Как можно и как нельзя? Иногда на собеседованиях просят найти ошибку в коде.

```
StudentChecks sc = (Student p) -> {return p.avgGrade > 8;};
info.testStudents(students, sc);
присвоить лямбда выражение переменной и дальше использовать
```

```
Collections.sort(students, (stud1, stud2) -> stud1.course-stud2.course);
System.out.println(...);
```

Can be replaced with Comparator.comparingInt more... (Ctrl+F1)

ArrayList students

Java понимает, что мы используем элементы типа Students (перезаписали метод Compare)

в параметре метода уже есть x и мы в теле метода опять задаём значение x

```
method((int x, int y) -> {int x=5; return10;});
 ↑
```

NOT OK

в параметре переданы x и y, а в теле метода присваивается значение - OK!

```
method((int x, int y) -> {x=5; return10;});
 ↑
```

OK

в теле метода лямбда выражения МОЖНО создавать новую переменную x2

```
method((int x, int y) -> {int x2=5; return10;});
 ↑
```

OK

Лямбда выражения работают с интерфейсом, в котором есть только 1 абстрактный метод. Такие интерфейсы называются функциональными интерфейсами, т.е. интерфейсами, пригодными для функционального программирования.

```
def(() -> 5);
```

Compile time errors:

```
def((x) -> x.length());
```

Поставил фигурные скобки? Пиши return!

```
def(x -> {x.length();});
 ↑
```

```
def((String x) -> x.length());
```

Поставили фиг. скобки и return, но забыли ;"

```
def(x -> {return x.length();});
 ↑
```

```
def((x, y) -> x.length());
```

Используете два параметра? Ставьте скобки!

```
def(x, y -> x.length());
 ↑
```

```
def((String x, String y) -> x.length());
```

## Stream API

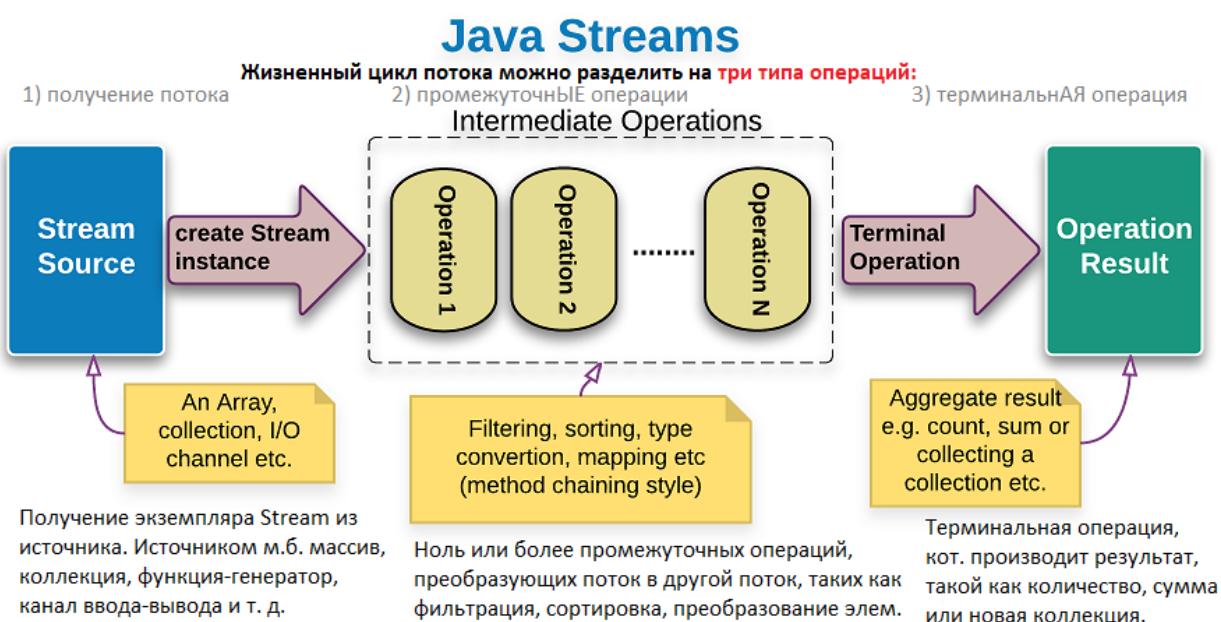
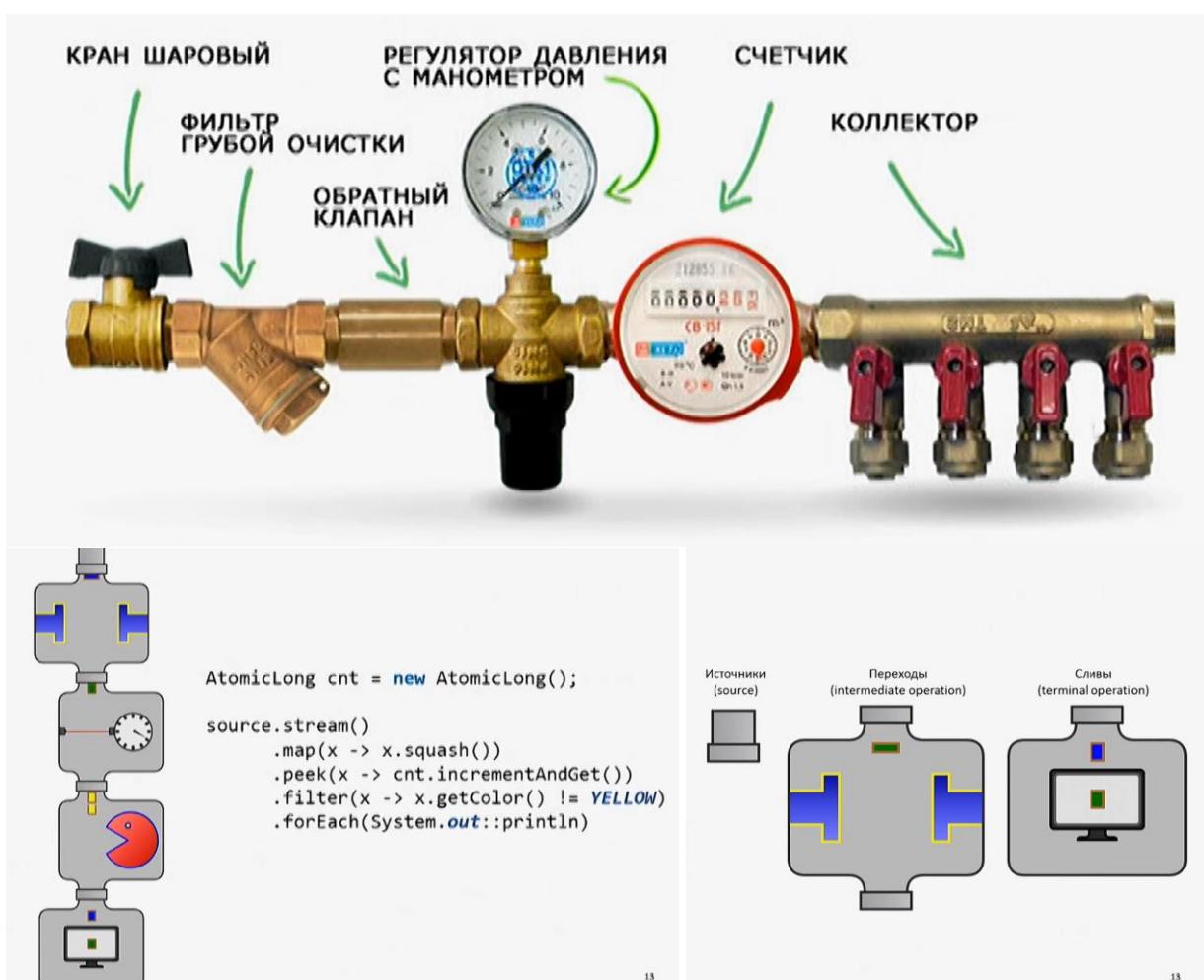
### 1. Что такое Stream API? Для чего нужны стримы? <https://youtu.be/RzEiCguFZiY>

Инструмент языка Java, который позволяет использовать функциональный стиль при работе с разными структурами данных (аналогия КОНВЕЙЕР).

Документация: Классы для поддержки операций функционального стиля над потоками элементов, таких как преобразование map-reduce в коллекциях.

```
int sum = widgets.stream()
 .filter(b -> b.getColor() == RED)
 .mapToInt(b -> b.getWeight())
 .sum();
```

Ключевой абстракцией, представленной в пакете **java.util.stream** является **ПОТОК**. Конструкция языка представляющая последовательность элементов, потенциально бесконечная, с возможностью применять к ней, сложные, поэтапные преобразования без цикла и условного оператора. Рекомендую видео Тагира Валеева <https://youtu.be/vxikpWnnnCU>



Stream не может быть использован повторно.  
 Как только была вызвана терминальная операция, stream закрывается.

**Какие бывают стримы?**

- 1)
- конечные и бесконечные;**

Бесконечные (`iterate`, `generate`) - создают бесконечный поток данных, до срабатывания условия.

- 2)
- последовательные и параллельные;**

Стримы бывают последовательными (`sequential`) и параллельными (`parallel`).

Последовательные выполняются только в текущем потоке, а вот параллельные используют общий пул `ForkJoinPool.commonPool()`.

При этом элементы разбиваются (если это возможно) на несколько групп и обрабатываются в каждом потоке отдельно.

Затем на нужном этапе группы объединяются в одну для предоставления конечного результата.

Чтобы получить параллельный стрим, нужно либо вызывать метод `parallelStream()` вместо `stream()`, либо превратить обычный стрим в параллельный, вызвав промежуточный оператор `parallel`.

- 3)
- объектные и примитивные;**

Специальные типы стримов для примитивных типов: `LongStream()`, `DoubleStream()`, `IntStream()`.

## 2. Почему Stream называют ленивым?

Ленивая инициализация - приём в разработке, когда ресурсоемкая операция выполняется только тогда, когда нужен ее результат. Для оптимизации производительности.

Stream API так и работает: мы указываем промежуточные операции, которые нам необходимо произвести и в конце следует завершающая или **терминальная операция, которая запускает все вычисления**.

**Ленивая обработка потоков**

Ленивая инициализация — это концепция отсрочки создания объекта до тех пор, пока объект не будет фактически впервые использован.

При правильном использовании это может привести к значительному повышению производительности.

Все промежуточные операции выполняются только тогда, когда есть терминальная операция.

Ленивая обработка потоков обеспечивает значительную эффективность; в конвейере, таком как приведенный выше пример `filter-map-sum`, фильтрация, сопоставление и суммирование могут быть объединены в один проход данных с минимальным промежуточным состоянием.

Также лень позволяет не проверять все данные, когда в этом нет необходимости; для таких операций, как «найти первую строку длиннее 1000 символов», из источника необходимо проверить только достаточное количество строк, чтобы найти строку с требуемыми характеристиками, не просматривая все строки, доступные из источника. Это поведение становится еще более важным, когда входной поток бесконечен, а не просто велик.

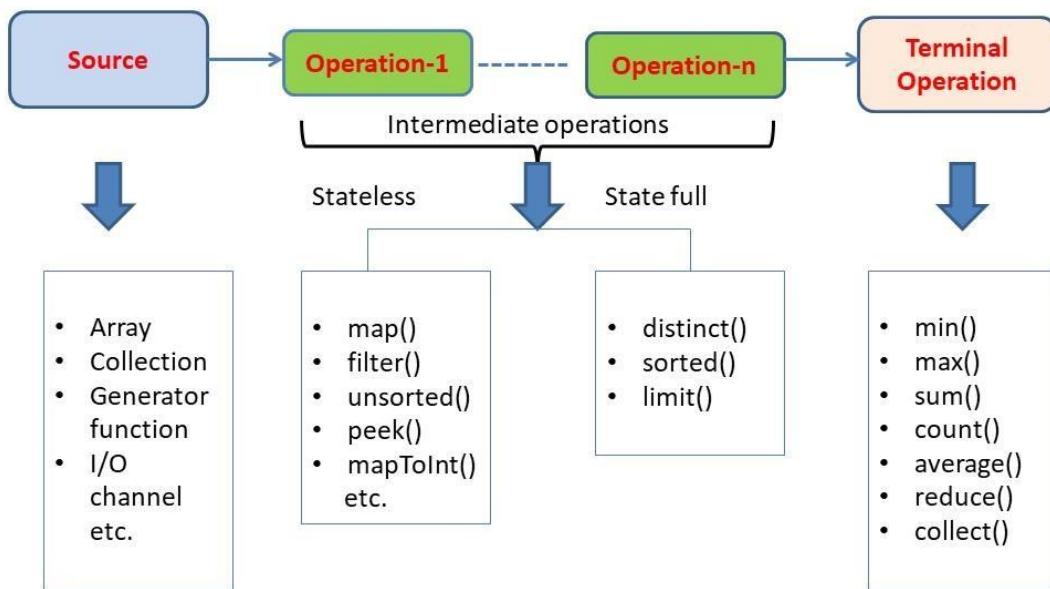
## 3. Какие существуют способы создания стрима?

| Способ создания                                                                        |                                                                     |                                                                                                                                                          |
|----------------------------------------------------------------------------------------|---------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1) Классический: Создание стрима из коллекции                                          | <code>collection.stream()</code>                                    | <code>Collection&lt;String&gt; collection = Arrays.asList("a1", "a2", "a3");<br/>Stream&lt;String&gt; streamFromCollection = collection.stream();</code> |
| 2) Создание стрима из значений                                                         | <code>Stream.of(значение1... значениеN)</code>                      | <code>Stream&lt;String&gt; streamFromValues = Stream.of("a1", "a2", "a3");</code>                                                                        |
| 3) Создание стрима из массива                                                          | <code>Arrays.stream(массив)</code>                                  | <code>String[] array = {"a1", "a2", "a3"};<br/>Stream&lt;String&gt; streamFromArray = Arrays.stream(array);</code>                                       |
| 4) Создание стрима из файла (каждая строка в файле будет отдельным элементом в стриме) | <code>Files.lines(путь_к_файлу)</code>                              | <code>Stream&lt;String&gt; streamFromFile = Files.lines(Paths.get("file.txt"))</code>                                                                    |
| 5) Создание стрима из строки                                                           | <code>«строка».chars()</code>                                       | <code>IntStream streamFromString = "123".chars()</code>                                                                                                  |
| 6) С помощью <code>Stream.builder</code>                                               | <code>Stream.builder().add(...).build()</code>                      | <code>Stream.builder().add("a1").add("a2").add("a3").build()</code>                                                                                      |
| 7) Создание параллельного стрима                                                       | <code>collection.parallelStream()</code>                            | <code>Stream&lt;String&gt; stream = collection.parallelStream();</code>                                                                                  |
| 8) Создание бесконечных стрима с помощью <code>Stream.iterate</code>                   | <code>Stream.iterate(начальное_условие, выражение_генерации)</code> | <code>Stream&lt;Integer&gt; streamFromIterate = Stream.iterate(1, n -&gt; n + 1)</code>                                                                  |
| 9) Создание бесконечных стрима с помощью <code>Stream.generate</code>                  | <code>Stream.generate(выражение_генерации)</code>                   | <code>Stream&lt;String&gt; streamFromGenerate = Stream.generate(() -&gt; "a1")</code>                                                                    |

## 4. Как из коллекции создать стрим?

```
Collection<String> collection = Arrays.asList("f5", "b6", "z7");
collection.stream();
```

## 5. Какие промежуточные методы в стримах вы знаете?



## 6. Методы КОНВЕЙЕРНЫЕ

| Метод                                                       | Что сделает                                                                           | Использование                                                                                                     |
|-------------------------------------------------------------|---------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| filter                                                      | отработает как фильтр, вернет значения, которые подходят под заданное условие         | collection.stream().filter(«e22»::equals).count()                                                                 |
| sorted                                                      | отсортирует элементы в естественном порядке; можно использовать Comparator            | collection.stream().sorted().collect(Collectors.toList())                                                         |
| limit                                                       | лимитирует вывод по тому, количеству, которое вы укажете                              | collection.stream().limit(10).collect(Collectors.toList())                                                        |
| skip                                                        | пропустит указанное вами количество элементов                                         | collection.stream().skip(3).findFirst().orElse("4")                                                               |
| distinct                                                    | найдет и уберет элементы, которые повторяются; вернет элементы без повторов           | collection.stream().distinct().collect(Collectors.toList())                                                       |
| peek                                                        | выполнит действие над каждым элементом элементов, вернет стрим с исходными элементами | collection.stream().map(String::toLowerCase).peek((e) -> System.out.print(", " + e)).collect(Collectors.toList()) |
| map                                                         | выполнит действия над каждым элементом; вернет элементы с результатами функций        | Stream.of("3", "4", "5").map(Integer::parseInt).map(x -> x + 10).forEach(System.out::println);                    |
| mapToInt<br>mapToDouble<br>mapToLong                        | Сработает как map, только вернет числовой stream                                      | collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()                                                |
| flatMap<br>flatMapToInt<br>flatMapToDouble<br>flatMapToLong | сработает как map, но преобразует один элемент в ноль, один или множество других      | collection.stream().flatMap((p) -> Arrays.asList(p.split(", ")).stream().toArray(String[])::                      |

## 7. Методы ТЕРМИНАЛЬНЫЕ

Метод **reduce()** выполняет функцию callback один раз для каждого элемента, присутствующего в массиве, за исключением пустот, принимая четыре аргумента: начальное значение (или значение от предыдущего вызова callback ), значение текущего элемента, текущий индекс и массив, по которому происходит итерация.

| Метод          | Что сделает                                                                           | Использование                                                                    |
|----------------|---------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| findFirst      | вернет элемент, соответствующий условию, который стоит первым                         | collection.stream().findFirst().orElse("10")                                     |
| findAny        | вернет любой элемент, соответствующий условию                                         | collection.stream().findAny().orElse("10")                                       |
| collect        | соберет результаты обработки в коллекции и не только                                  | collection.stream().filter((s) -> s.contains("10")).collect(Collectors.toList()) |
| count          | посчитает и выведет, сколько элементов, соответствующих условию                       | collection.stream().filter("f5":equals).count()                                  |
| anyMatch       | True, когда хоть один элемент соответствует условиям                                  | collection.stream().anyMatch("f5":equals)                                        |
| noneMatch      | True, когда ни один элемент не соответствует условиям                                 | collection.stream().noneMatch("b6":equals)                                       |
| allMatch       | True, когда все элементы соответствуют условиям                                       | collection.stream().allMatch((s) -> s.contains("8"))                             |
| min            | найдет самый маленький элемент, используя переданный сравнитель                       | collection.stream().min(String::compareTo).get()                                 |
| max            | найдет самый большой элемент, используя переданный сравнитель                         | collection.stream().max(String::compareTo).get()                                 |
| forEach        | применит функцию ко всем элементам, но порядок выполнения гарантировать не может      | set.stream().forEach((p) -> p.append("_2"));                                     |
| forEachOrdered | применит функцию ко всем элементам по очереди, порядок выполнения гарантировать может | list.stream().forEachOrdered((p) -> p.append("_ny"));                            |
| toArray        | приведет значения стрима к массиву                                                    | collection.stream().map(String::toLowerCase).toArray(String[]::new);             |

## 8. Методы числовых стримов

Это специальные методы, которые работают только со стримами с числовыми примитивами.

| Метод    | Что сделает                                    | Использование                                                      |
|----------|------------------------------------------------|--------------------------------------------------------------------|
| sum      | вернет сумму чисел, представленных в коллекции | collection.stream().mapToInt((s) -> Integer.parseInt(s)).sum()     |
| average  | вернет среднее арифметическое                  | collection.stream().mapToInt((s) -> Integer.parseInt(s)).average() |
| mapToObj | преобразует числовой стрим в объектный         | intStream.mapToObj((id) -> new Key(id)).toArray()                  |

## 9. Расскажите про класс Collectors и его методы. <https://www.youtube.com/watch?v=HOQpQWK9u28>

```
public final class Collectors extends Object
```

Реализации интерфейса Collector, реализующие различные полезные операции редукции, такие как накопление элементов в коллекции, суммирование элементов по различным критериям и т. д. Методы этого класса значительно упрощают использование **аккумулирующих терминальных методов**. Рассмотрим методы по группам.

**Методы для сбора коллекций.** Предназначены для сбора элементов потока в ту или иную реализацию интерфейса Collector.

| Метод                                                                                              | Описание                                         |
|----------------------------------------------------------------------------------------------------|--------------------------------------------------|
| static <T,C extends Collection<T>><br>Collector<T,?,C> toCollection(Supplier<C> collectionFactory) | Собирает элементы потока в реализацию Collection |
| static <T> Collector<T,?,List<T>> toList()                                                         | Собирает элементы потока в List                  |
| static <T> Collector<T,?,Set<T>> toSet()                                                           | Собирает элементы потока в Set                   |
| static <T> Collector<T,?,List<T>> toUnmodifiableList()                                             | Собирает элементы в неизменяемый List            |
| static <T> Collector<T,?,Set<T>> toUnmodifiableSet()                                               | Собирает элементы в неизменяемый Set             |

```

Main.java
1 package sample1;
2
3 import java.util.List;
4
5
6 public class Main {
7
8 public static void main(String[] args) {
9
10 List<Integer> listNumber = List.of(1, 2, 3, 4, 5);
11
12 List<Integer> result = listNumber.stream()
13 .filter(a -> a%2==0)
14 .collect(Collectors.toList());
15
16 System.out.println(result);
17
18 }
19
20 }

```

<terminated> Main (128) [Java Application] /home/alexander/Документы, [2, 4]

## Методы для сбора элементов потока в Map

Три вида Map можем получить:

- Обычную изменяемую карту (первые три метода)
- Неизменяемая карта (следующие два метода)
- Поток безопасная карта (следующие три метода)

| Метод                                                                                                                                                                                                                             | Описание                                                                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static <T,K,U>Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)                                                                                               | Собирает элементы потока в Map применяя keyMapper для генерации ключа и valueMapper для генерации значений                                                                                                                                                   |
| static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)                                                             | Собирает элементы потока в Map применяя keyMapper для генерации ключа и valueMapper для генерации значений. Для слияния эквивалентных результатов используют mergeFunction                                                                                   |
| static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapFactory)                         | Собирает элементы потока в Map применяя keyMapper для генерации ключа и valueMapper для генерации значений. Для слияния эквивалентных результатов используют mergeFunction. В какую именно карту собирать результат определяется mapFactory                  |
| static <T,K,U>Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)                                                                                   | Собирает элементы потока в неизменяемый Map применяя keyMapper для генерации ключа и valueMapper для генерации значений.                                                                                                                                     |
| static <T,K,U>Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)                                                  | Собирает элементы потока в неизменяемый Map применяя keyMapper для генерации ключа и valueMapper для генерации значений. Для слияния эквивалентных результатов используют mergeFunction                                                                      |
| static <T,K,U>Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)                                                                           | Собирает элементы потока в потокобезопасную Map применяя keyMapper для генерации ключа и valueMapper для генерации значений                                                                                                                                  |
| static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)                                         | Собирает элементы потока в потокобезопасный Map применяя keyMapper для генерации ключа и valueMapper для генерации значений. Для слияния эквивалентных результатов используют mergeFunction                                                                  |
| static <T,K,U,M extends ConcurrentHashMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapFactory) | Собирает элементы потока в потокобезопасный Map применяя keyMapper для генерации ключа и valueMapper для генерации значений. Для слияния эквивалентных результатов используют mergeFunction. В какую именно карту собирать результат определяется mapFactory |

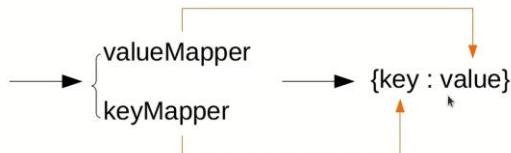
```

Main.java
1 package sample3;
2
3 import java.util.ArrayList;
4
5
6 public class Main {
7
8 public static void main(String[] args) {
9
10 List<Integer> listNumber = List.of(1, 2, 3, 4, 5);
11
12 Function<Integer, String> keyMapper = a -> (a % 2 == 0) ? "even" : "odd";
13
14 Function<Integer, List<Integer>> valueMapper = a -> List.of(a);
15
16 BinaryOperator<List<Integer>> mergeFunction = (a, b) -> {
17 List<Integer> result = new ArrayList<>(a);
18 result.addAll(b);
19 return result;
20 };
21
22 Supplier<Map<String, List<Integer>>> supplier = HashMap::new;
23
24 Map<String, List<Integer>> result = listNumber.stream()
25 .collect(Collectors.toMap(keyMapper, valueMapper, mergeFunction, supplier));
26
27 System.out.println(result);
28
29 }
30
31 }

```

<terminated> Main (130) [Java Application] /home/alexander/Документы/Install/GraalVM {even=[2, 4], odd=[1, 3, 5]}

## Как работает метод toMap



Этот метод возвращает реализацию Collector собирающий элементы потока в карту. С помощью keyMapper создаются ключи на основании элементов потока, каждому созданному ключу соответствует значение генерируются с помощью valueMapper на основании элемента потока.

### Методы для группировки данных в Map

| Метод                                                                                                                                                                                        | Описание                                                                                                                                                                                        |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)                                                                                            | Проводит операцию группировки с помощью classifier в Map                                                                                                                                        |
| static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)                     | Проводит операцию группировки в Map с помощью classifier генерируя ключи карты (создается mapFactory). В качестве значения используется результат работы коллектора downstream                  |
| static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)                                                         | Проводит операцию группировки в Map с помощью classifier генерируя ключи карты. В качестве значения используется результат работы коллектора downstream                                         |
| static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)                                                                        | Проводит операцию группировки с помощью classifier в потокобезопасный Map                                                                                                                       |
| static <T,K,A,D,M extends ConcurrentMap<K,D>> Collector<T,?,M> groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream) | Проводит операцию группировки в потокобезопасный Map с помощью classifier генерируя ключи карты (создается mapFactory). В качестве значения используется результат работы коллектора downstream |
| static <T,K,A,D> Collector<T,?,ConcurrentMap<K,D>> groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)                                     | Проводит операцию группировки в потокобезопасный Map с помощью classifier генерируя ключи карты. В качестве значения используется результат работы коллектора downstream                        |

### Методы для изменения методов сбора

| Метод                                                                                                                                        | Описание                                                                                                                                                             |
|----------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)                                  | Возвращает Collector на основании Collector выступающего в качестве параметра к которому применена Function                                                          |
| static <T,A,R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)                               | Вернет Collector на основании Collector выступающего в качестве параметра, который соберет только те элементы потока для которых Predicate вернет true               |
| static <T,U,A,R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream) | Вернет Collector на основании Collector выступающего в качестве параметра, который собирает элементы потоков созданных с помощью Function на основании данных потока |

### Методы для получения статистики

| Метод                                                                                                   | Описание                                                                                                               |
|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| static <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper)                    | Вернет Collector который вычисляет среднеарифметическое элементов, полученных из потока с помощью mapper               |
| static <T> Collector<T,?,Double> averagingInt(TolntFunction<? super T> mapper)                          | Вернет Collector который вычисляет среднеарифметическое элементов, полученных из потока с помощью mapper               |
| static <T> Collector<T,?,Double> averagingLong(ToLongFunction<? super T> mapper)                        | Вернет Collector который вычисляет среднеарифметическое элементов, полученных из потока с помощью mapper               |
| static <T> Collector<T,?,Long> counting()                                                               | Вернет Collector считающий количество элементов в потоке                                                               |
| static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)                      | Вернет Collector который вычисляет сумму элементов, полученных из потока с помощью mapper                              |
| static <T> Collector<T,?,Integer> summingInt(TolntFunction<? super T> mapper)                           | Вернет Collector который вычисляет сумму элементов, полученных из потока с помощью mapper                              |
| static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)                            | Вернет Collector который вычисляет сумму элементов, полученных из потока с помощью mapper                              |
| static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper) | Вернет Collector возвращающий сборную статистику полученную на основании элементов потока, созданного с помощью mapper |
| static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(TolntFunction<? super T> mapper)          | Вернет Collector возвращающий сборную статистику полученную на основании элементов потока, созданного с помощью mapper |
| static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)       | Вернет Collector возвращающий сборную статистику полученную на основании элементов потока, созданного с помощью mapper |
| static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)                           | Вернет Collector возвращающий Optional с максимальных элементов потока                                                 |
| static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)                           | Вернет Collector возвращающий Optional с минимальным элементов потока                                                  |

Main.java

```
1 package sample13;
2
3 import java.util.List;
4
5 public class Main {
6
7 public static void main(String[] args) {
8 List<String> names = List.of("Katia", "Alexander", "Olga", "Inna");
9 String result = names.stream().collect(Collectors.joining(", ", "Hello ", ""));
10 System.out.println(result);
11 }
12
13 }
```

Console

```
<terminated> Main (140) [Java Application] /home/alexander/Документы/Install/Gr Hello Katia, Alexander, Olga, Inna!
```

### Задание для самостоятельной работы:

- 1) Соберите элементы потока целых чисел в две строки (одна для четных чисел, вторая для нечетных чисел) в качестве разделятеля используйте «;».
- 2) Реализуйте файловый классификатор по размеру файлов.

## 10. Расскажите о параллельной обработке в Java 8.

В Java бывают еще и параллельные стримы, обрабатывающие свои элементы одновременно в нескольких потоках.

Рассмотрим несколько полезных методов, которые помогают управлять последовательными и параллельными стримами — как минимум быстро их определять.

| Метод      | Что делает                                         | Использование                    |
|------------|----------------------------------------------------|----------------------------------|
| isParallel | скажет, параллельный стрим или нет                 | someStream.isParallel()          |
| parallel   | сделает стрим параллельным или вернет сам себя     | someStream = stream.parallel()   |
| sequential | сделает стрим последовательным или вернет сам себя | someStream = stream.sequential() |

Стримы могут быть последовательными и параллельными. Первые выполняются в текущем потоке, вторые используют общий пул **ForkJoinPool.commonPool()**

В параллельном стриме элементы разделяются на группы. Их обработка проходит в каждом потоке по отдельности. Затем они снова объединяются, чтобы вывести результат. С помощью методов `parallel` и `sequential` можно явно указать, что нужно сделать параллельным, а что — последовательным.

**Не рекомендуется применять параллельность для выполнения долгих операций** (например, извлечения данных из базы), потому что все стримы работают с общим пулом. Долгие операции могут остановить работу всех параллельных стримов в Java Virtual Machine из-за того, что в пуле не останется доступных потоков.

Чтобы избежать такой проблемы, используйте параллельные стримы **только для коротких операций**, выполнение которых занимает миллисекунды, а не секунды и тем более минуты.

В Stream API по умолчанию скрыта работа с поток небезопасными коллекциями, разделение на части и объединение элементов. Это отличное решение. Разработчику остается только выбирать нужные методы и следить за тем, чтобы не было зависимостей от внешних факторов.

## 11. Что такое IntStream и DoubleStream?

Существуют специализированные stream-ы: IntStream, LongStream, DoubleStream.

У них есть все методы, что и в обычном Stream, но также существуют дополнительные методы: **count, average, sum, min, max, range** и другие.

Эти stream-ы **специально созданы для примитивных типов**, так как обычный Stream работает с объектами, а значит, в нем будут накладные расходы на автоупаковку и автораспаковку. Специализированные stream-ы есть только для int, long и double. Для других примитивных типов специализированных stream-ов нет.

Пример использования IntStream:

```
1 import java.util.OptionalDouble;
2 import java.util.stream.IntStream;
3
4 public class AverageMain {
5 public static void main(String[] args) {
6 IntStream intStream = IntStream.builder().add(10).add(14).add(-5)
7 .add(33).build();
8
9 OptionalDouble average = intStream.average();
10 average.ifPresentOrElse(v -> System.out.println("average = " + v),
11 () -> System.out.println("Stream was empty."));
12 }
13 }
14 }
```

Примеры использования DoubleStream:

```
1 import java.util.stream.DoubleStream;
2
3 public class DoubleMain {
4 public static void main(String[] args) {
5 DoubleStream doubleStream = DoubleStream.builder()
6 .add(10.0)
7 .add(14.4)
8 .add(-5.3)
9 .add(33.8)
10 .build();
11
12 double sum = doubleStream.sum();
13 System.out.println("sum = " + sum);
14 }
15 }
16 }
```

Java 8 и более старшие версии

### 1. Какие нововведения появились в java 8?

- Методы интерфейсов по умолчанию;
- Лямбда-выражения;
- Функциональные интерфейсы;
- Ссылки на методы и конструкторы;

- Повторяемые аннотации;
- Аннотации на типы данных;
- Рефлексия для параметров методов;
- **Stream API для работы с коллекциями;**
- Параллельная сортировка массивов;
- **Новое API для работы с датами и временем;**
- Новый движок JavaScript *Nashorn*;
- Добавлено несколько новых классов для потокобезопасной работы;
- Добавлен новый API для Calendar и Locale;
- Добавлена поддержка *Unicode 6.2.0*;
- **Добавлен стандартный класс для работы с Base64;**
- Добавлена поддержка беззнаковой арифметики;
- Улучшена производительность конструктора `java.lang.String(byte[], *)` и метода `java.lang.String.getBytes()`;
- Новая реализация `AccessController.doPrivileged`, позволяющая устанавливать подмножество привилегий без необходимости проверки всех остальных уровней доступа;
- *Password-based* алгоритмы стали более устойчивыми;
- Добавлена поддержка *SSL/TLS Server Name Indication (NSI)* в JSSE Server;
- Улучшено хранилище ключей (KeyStore);
- Добавлен алгоритм *SHA-224*;
- Удален мост *JDBC - ODBC*;
- Удален *PermGen*, изменен способ хранения мета-данных классов;
- Возможность создания профилей для платформы Java SE, которые включают в себя не всю платформу целиком, а некоторую ее часть;
- Инструментарий
  - Добавлена утилита *jjs* для использования *JavaScript Nashorn*;
  - Команда *java* может запускать *JavaFX* приложения;
  - Добавлена утилита *jdeps* для анализа *.class*-файлов.

## 2. Нововведения Java 17 (посмотреть видео по ссылке)

[https://www.youtube.com/watch?v=GL6\\_Fc53tDw](https://www.youtube.com/watch?v=GL6_Fc53tDw)

### Что может сломаться при переходе на 17 версию

- Finalizers removed from some methods
  - `java.io.FileInputStream`
  - `java.util.zip.Deflater/Inflater/ZipFile`
  - `java.awt.color.ICC_Profile`
- Package `java.util.jar`: Interfaces `Packer` and `Unpacker`, implementing class `Pack200` removed
- Constructor in `java.net.URLDecoder` removed
- A number of AWT and Swing classes changed visibility attributes and received other cleanup
- Deprecated for removal:
  - Applet API

### Breaking Changes - Security and VM

Security & cryptography

- Different root certificates
- Removal of NIST EC Curves from the default TLS Algorithms
- Disabled Native SunEC Implementation by Default
- Deprecated for removal: `Security manager`
- Removed `java.security.acl` & `java.rmi.activation` packages



VM

- GraalVM JIT no longer bundled with JDK
- CMS GC removed
- Biased locking removed (impacting, not breaking)

### Key JVM Changes

Lot's of small changes under the hood

- GC
- CMS is gone
- ZGC is very good and production ready
  - But don't use ZGC in JDK 11
- Unused memory returned to the OS (G1, earlier in Shenandoah)
- Abortable Mixed Collections for G1
- NUMA-Aware Memory Allocation for G1
- More concurrent ops => smaller GC pauses for all GCs

CMS выпилили, его в 17й Java нет  
По дефолту JVM использует Garbage First G1

Несколько слов про ZGC

Экспериментально появился в Java 11 (НО!!! Не используйте там, есть баги, которые никто не чинит)  
это хороший коллектор, он стал продуктивным в Java 15, в Java 17 он совсем хороший

Shenandoah

GC он такой жадный, сначала пытается подобрать под сея побольше памяти, а потом ее использовать  
Любому GC нужно дополнительное место в памяти, чтобы копировать, потом уплотнять и т.д.  
А в облачной среде, когда приходится платить за память, которая висит за процессом,  
есть финансовый смысл, чтобы эту память обратно возвращать системе (чтобы Foot Print был поменьше)  
И такое появилось и в G1 и в Shenandoah

#### Garbage First G1

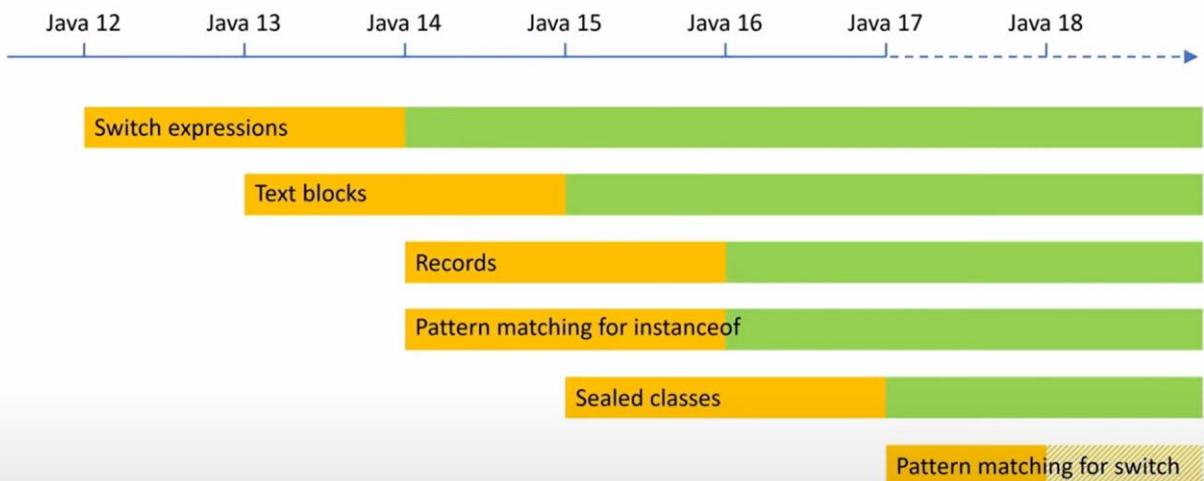
Появилось Abortable Mixed Collections. Когда у нас идет сборка мусора, то G1 разбили уборку по приоритетам.  
Часть регионов собирается безусловно (т.е. в этом цикле должны быть собраны), а другие (например 20%)  
могут быть собраны позже, если текущий цикл не укладывается в отведенный временной бюджет  
Это шаг в сторону Real Time, уменьшения времени  
Также G1 научили алокировать память (сборщик мусора отвечает за выделение памяти и создание объектов)  
Выделяется память, область которой находится ближе к тред-исполнителю.

Раньше это было в Parallel GC, а теперь появилась и G1. Ну и много других маленьких вещей, нацеленных на процесс, параллельный с основным приложением (минимальные паузы).

<https://www.youtube.com/watch?v=1WaQoBiloK4>

## Изменения в языке

# JEP 12: Preview Features



<https://openjdk.java.net/jeps/12>

22

Record – семантически богатая штука и плохая новость для тех, кому платили за строчки кода - их стало меньше))  
В круглых скобках компоненты, в фигурных – заголовок

## Records

- ✓ Иммутабельны
- ✓ Всегда имеют канонический конструктор с параметрами, соответствующими компонентам
- ✓ Всегда имеет аксессоры с именами, соответствующими компонентам
- ✓ Два рекорда, сконструированные с одинаковыми параметрами, равны по equals и имеют равный hashCode
- ✓ Если считать компоненты через аксессоры и создать из них новый рекорд, он будет равен исходному по equals и иметь такой же hashCode

### Records – reflection

```
public record Point(int x, int y) {
 public static void main(String[] args) {
 boolean record = Point.class.isRecord();
 if (record) {
 RecordComponent[] components = Point.class.getRecordComponents();
 System.out.println(Arrays.toString(components));
 Method accessor = components[0].getAccessor();
 System.out.println(accessor);
 }
 }
 [int x, int y]
 public int Point.x()
}
```

get! Смотрите, get!

```
public record Point(int x, int y) implements Serializable {
 public static void main(String[] args) throws IOException {
 Point point = new Point(-1, -1);
 var result = new ByteArrayOutputStream();
 try (var oos = new ObjectOutputStream(result)) {
 oos.writeObject(point);
 }
 System.out.println(Base64.getEncoder().encodeToString(result.toByteArray()));
 }
}
r00ABXNyAAVQb2LudAAAAAAAAGACSQABeEKAAL4cP/////////
```

### Sealed classes

### Sealed classes

```
Java 15: Preview
Java 17: Standard

public abstract sealed class Pet permits Cat, Dog, Parrot, Goldfish {
 ...
}
```

```
public abstract sealed class Pet permits Cat, Dog, Parrot, Goldfish {
 ...
}

public sealed class Dog extends Pet {
 public static final class Dachshund extends Dog {}
 public static final class ChowChow extends Dog {}
 public static final class Collie extends Dog {}
}

public final class Goldfish extends Pet {}
public final class Parrot extends Pet {}
public non-sealed class Cat extends Pet {}
```

## Stream.teeing (Java 12)

```
Collector<BigDecimal, ?, BigDecimal> averaging =
 Collectors.teeing(
 // collector1
 Collectors.reducing(BigDecimal.ZERO, BigDecimal::add),
 // collector2
 Collectors.counting(),
 // merger
 (sum, count) -> sum.divide.BigDecimal.valueOf(count),
 2, RoundingMode.HALF_EVEN);

BigDecimal average = Stream.of(BigDecimal.ONE, BigDecimal.TEN)
 .collect(averaging);
System.out.println(average); // 5.50
```

## interface RandomGenerator (Java 17)

```
DoubleStream doubles(...)
IntStream ints(...)
LongStream longs(...)
boolean nextBoolean()
void nextBytes(byte[])
float nextFloat(...)
double nextDouble(...)
int nextInt(...)
long nextLong(...)
double nextGaussian(...)
double nextExponential()
```

### 3. Какие новые классы для работы с датами появились в java 8?

- LocalDate – дата без времени и временных зон;
- LocalTime – время без даты и временных зон;
- LocalDateTime – дата и время без временных зон;
- ZonedDateTime – дата и время с временной зоной;
- DateTimeFormatter – форматирует даты в строки и наоборот, только для классов java.time;

### 4. Расскажите про класс Optional

Опциональное значение `Optional` — это контейнер для объекта, который может содержать или не содержать значение `null`. Такая обёртка является удобным средством предотвращения `NullPointerException`, т.к. имеет некоторые функции высшего порядка, избавляющие от добавления повторяющихся `if null/notNull` проверок:

```
Optional<String> optional = Optional.of("hello");

optional.isPresent(); // true
optional.ifPresent(s -> System.out.println(s.length())); // 5
optional.get(); // "hello"
optional.orElse("ops..."); // "hello"
```

`package java.util`

`public final class Optional <T>` - это мини-контейнер для одного единственного объекта произвольного типа. При этом объект опционален, то есть может отсутствовать.

**Зачем нужен этот класс?** Применяя `String`, нужно делать проверку на `null` или явно указывать аннотацию (сторонняя библиотека). `Optional` решает эту проблему.

```
String text = ???; // is null allowed?

@Nullable String nullableText = null;

@NonNull String nonNullText = "hello!";

String text = "bar";

Optional<String> optionalText =
 Optional.of("baz");
```

```
package java.util;

public final class Optional<T> {

 private final T value;

 private Optional(T value) {
 this.value = Objects.requireNonNull(value);
 }

 public static <T> Optional<T> of(T value) {
 return new Optional<>(value);
 }

 public T get() {
 if (value == null) {
 throw new NoSuchElementException("No value present");
 }
 return value;
 }

 // ...
}
```

```
Optional<String> baz = Optional.of("baz");
baz.ifPresent(System.out::println);
if (s != null) {
 System.out.println(s);
}
String value = bar.orElse("bar");
s!=null ? s : "bar"
```

## 5. Что такое Nashorn?

**Nashorn** – это движок JavaScript, разрабатываемый на Java компанией Oracle.

Призван дать возможность встраивать код JavaScript в приложения Java. В сравнении с *Rhino*, который поддерживается Mozilla Foundation, Nashorn обеспечивает от 2 до 10 раз более высокую производительность, так как он компилирует код и передает байт-код виртуальной машине Java непосредственно в память.

Nashorn умеет компилировать код JavaScript и генерировать классы Java, которые загружаются специальным загрузчиком. Так же возможен вызов кода Java прямо из JavaScript.

## 6. Что такое jjs?

jjs это утилита командной строки, которая позволяет выполнять программы на языке JavaScript прямо в консоли.

## 7. Какой класс появился в Java 8 для кодирования/декодирования данных?

Base64 – потокобезопасный класс, который реализует кодировщик и декодировщик данных, используя схему кодирования base64 согласно RFC 4648 и RFC 2045.

Base64 содержит 6 основных методов:

```
getEncoder() / getDecoder() - возвращает кодировщик/декодировщик base64, соответствующий стандарту RFC 4648;
getUrlEncoder() / getUrlDecoder() - возвращает URL-safe кодировщик/декодировщик base64, соответствующий стандарту RFC 4648;
getMimeEncoder() / getMimeDecoder() - возвращает MIME кодировщик/декодировщик, соответствующий стандарту RFC 2045.
```

## 7. Как создать Base64 кодировщик и декодировщик?

```
// Encode
String b64 = Base64.getEncoder().encodeToString("input".getBytes("utf-8")); //aW5wdXQ==
// Decode
new String(Base64.getDecoder().decode("aW5wdXQ=="), "utf-8"); //input
```

## 8. Какие дополнительные методы для работы с ассоциативными массивами (maps) появились в Java 8? <https://github.com/enhorse/java-interview/blob/master/java8.md>

- `putIfAbsent()` добавляет пару «ключ-значение», только если ключ отсутствовал:

```
map.putIfAbsent("a", "Aa");
```

- `forEach()` принимает функцию, которая производит операцию над каждым элементом:

```
map.forEach((k, v) -> System.out.println(v));
```

- `compute()` создаёт или обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

```
map.compute("a", (k, v) -> String.valueOf(k).concat(v)); //["a", "aAa"]
```

- `computeIfPresent()` если ключ существует, обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):

```
map.computeIfPresent("a", (k, v) -> k.concat(v));
```

- `computeIfAbsent()` если ключ отсутствует, создаёт его со значением, которое вычисляется (возможно использовать ключ):

```
map.computeIfAbsent("a", k -> "A".concat(k)); //["a", "Aa"]
```

- `getOrDefault()` в случае отсутствия ключа, возвращает переданное значение по-умолчанию:

```
map.getOrDefault("a", "not found");
```

- `merge()` принимает ключ, значение и функцию, которая объединяет передаваемое и текущее значения. Если под заданным ключем значение отсутствует, то записывает туда передаваемое значение.

```
map.merge("a", "z", (value, newValue) -> value.concat(newValue)); //["a", "Aaz"]
```

## 9. Что такое LocalDateTime?

`LocalDateTime` объединяет вместе `LocaleDate` и `LocalTime`, содержит дату и время в календарной системе ISO-8601 без привязки к часовому поясу.

Время хранится с точностью до наносекунды. Содержит множество удобных методов, таких как `plusMinutes`, `plusHours`, `isAfter`, `toSecondOfDay` и т.д.

## 10. Что такое ZonedDateTime?

`java.time.ZonedDateTime` — аналог `java.util.Calendar`, класс с самым полным объемом информации о временном контексте в календарной системе ISO-8601.

Включает временную зону, поэтому все операции с временными сдвигами этот класс проводит с её учётом.

\*Коллекции бывают разные: что такое Guava и с чем её едят? <https://github.com/google/guava>

## Что такое Guava?

По сути — это набор основных библиотек Google для Java с открытым исходным кодом.

Внутри можно встретить библиотеку графов, новые типы коллекций (`multiset` etc.), различные функциональные типы, богатый набор утилит для параллелизма, обработки строк, хэширования.

- использовать `firstNonNull` из класса `Objects` библиотеки Guava вместо записи “`if else`”
- использовать `Object.equals(a, b)` для безопасной проверки на равенство.
- использовать `Multimap` вместо `Map` со значениями `List` или `Set`.
- использовать `Multiset` для подсчета количества вхождений объекта.

<https://kataacademy.medium.com/%D1%87%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-quava-%D0%B8%D1%81%D1%87%D0%B5%D0%BC%D0%B5%D1%91-%D0%B5%D0%B4%D1%8F%D1%82-9f7b088ed243>