

COLLECTIONS FRAMEWORK, CRUD И ТЕСТИРОВАНИЕ СИСТЕМ, УПРАВЛЯЮЩИХ НАБОРОМ ОБЪЕКТОВ

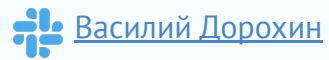


ВАСИЛИЙ ДОРОХИН



ВАСИЛИЙ ДОРОХИН

QuadCode, QA Engineer





План занятия

1. [Collections Framework](#)
2. [Collection](#)
3. [Wildcards](#)
4. [List](#)
5. [Default](#)
6. [Advanced](#)
7. [CRUD](#)
8. [Set](#)
9. [Map](#)
10. [Итоги](#)



COLLECTIONS FRAMEWORK



ПРОБЛЕМА

Мы с вами уже разобрали разделение обязанностей на уровни бизнес-логики, хранения данных и сами структуры хранения (массивы).

Но с массивами всё очень сложно:

1. Много дублирующегося кода для поиска/добавления/удаления
2. Постоянные риски выйти за границы с последующими исключениями и т.д.

Неужели нет какого-то способа попроще?



COLLECTIONS FRAMEWORK

Хранение набора объектов — это настолько частая задача, что разработчики стандартной библиотеки Java постарались и предоставили нам Collections Framework.

Collections Framework — это набор интерфейсов, классов с реализацией и готовых алгоритмов, обеспечивающих удобную работу с набором объектов.



COLLECTIONS FRAMEWORK

Collections Framework целиком построен на идее использования интерфейсов и generic'ов (именно поэтому мы сначала прошли их и только сейчас переходим к самому фреймворку).



ВАЖНО

Collections Framework развивался достаточно долгое время, и многие идеи с первого взгляда могут показаться непонятными и сложными.

Поэтому, относитесь к изучению этого фреймворка как обычно:

1. Сначала учимся использовать на том уровне понимания, который есть.
2. Затем глубже разбираемся с тем, что внутри.
3. Повторяем, начиная с п.1.



ВАЖНО

Что нам нужно знать о большей части того, что мы сегодня будем рассматривать:

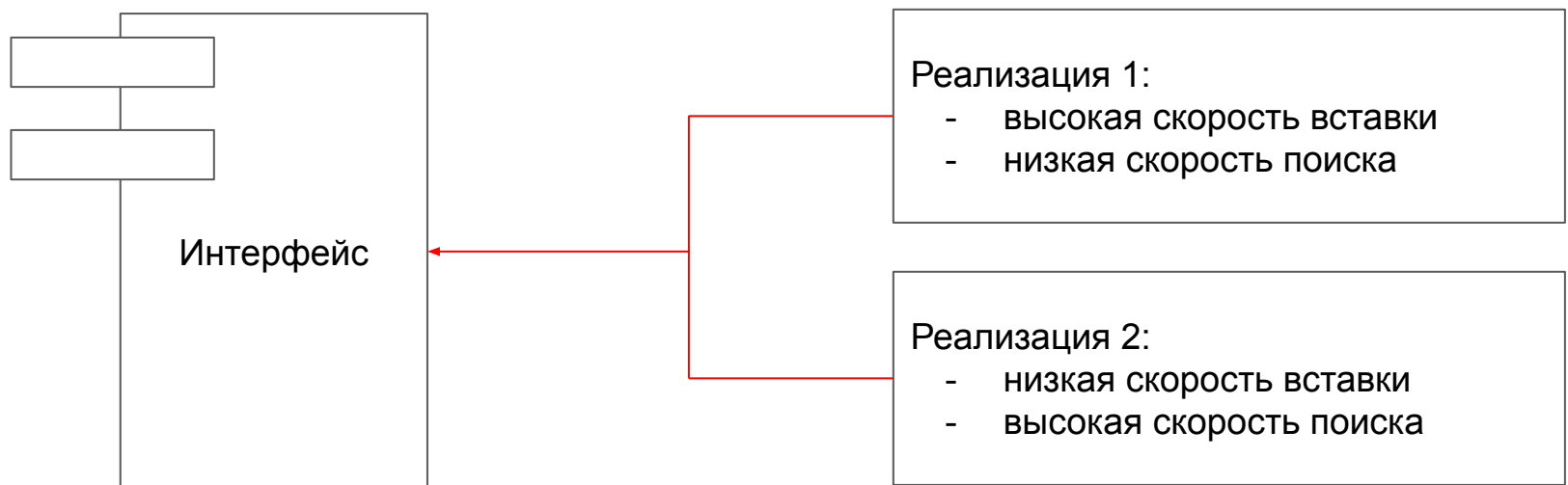
1. Что это существует.
2. Зачем это нужно.
3. Как использовать.

Сами вы такие конструкции с нуля писать либо не будете, либо мы их детально будем разбирать по мере использования в рамках курса «Автоматизация Тестирования».

КЛЮЧЕВАЯ ИДЕЯ

Ключевая идея в рамках Collections Framework состоит в разделении интерфейсов (требований к наличию методов) и конкретных реализаций.

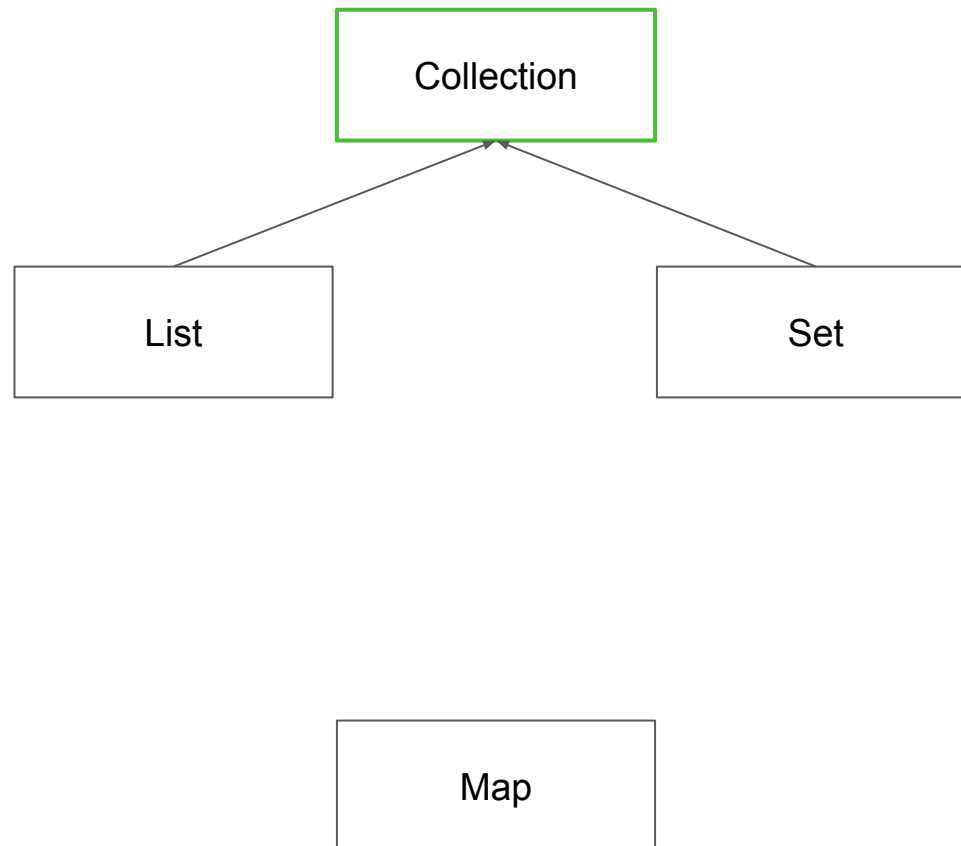
Реализации могут отличаться по скорости, потреблению памяти и другим характеристикам:





COLLECTION

КЛЮЧЕВЫЕ ИНТЕРФЕЙСЫ





COLLECTION

В этой лекции мы детально рассмотрим Collection и List, обзорно поговорим про остальные коллекции.

COLLECTION

// представлены только ключевые методы

```
public interface Collection<E> extends Iterable<E> {  
    int size();  
    boolean isEmpty();  
    boolean contains(Object o);  
    boolean add(E e);  
    boolean remove(Object o);  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    void clear();  
}
```

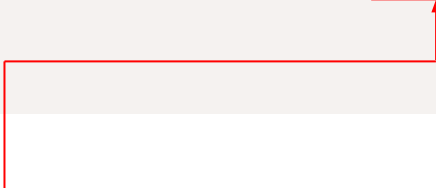
Имеют значение только первые угловые скобки, дальше везде E просто заменяется на нужный тип

Коллекция — это набор объектов, у которого есть:

1. Размер
2. Возможность добавлять/удалять элементы
3. Проверка на существование элемента
4. Возможность добавлять/удалять другие коллекции

НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ

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



Q: `extends` означает наследование классов. Что оно значит по отношению к интерфейсам?

A: в интерфейсах `extends` тоже означает наследование, но только это значит, что должны быть имплементированы все методы, описанные в текущем интерфейсе (`Collection`) + все методы, описанные в том интерфейсе, от которого наследуемся (`Iterable`).



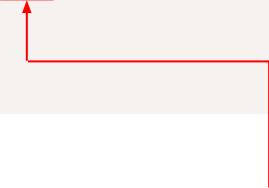
ITERABLE

Q: Что такое `Iterable`?

A: Это интерфейс, описывающий нечто, по чему можно «итерироваться». Для нас — это то, что можно перебрать в цикле `for-each` так же, как мы перебирали массив.

GENERIC

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



Q: Почему используется буква **E**, а не **T**?

A: При использовании generic'ов в отношении некоторых букв приняты общие соглашения (в Collections Framework):

- E — element (элемент)
- K — key (ключ)
- V — value (значение)
- T — type (тип)

Поскольку Collection описывает набор элементов, то используется **E**

COLLECTION

```
public interface Collection<E> extends Iterable<E> {  
    ...  
    int size();  
    boolean isEmpty();  
    void clear();  
    ...  
}
```

Методы:

1. `size` — количество элементов в коллекции.
2. `isEmpty` — пустая коллекция (true) или содержит элементы (false).
3. `clear` — очистка коллекции (удаление всех элементов).

COLLECTION

```
public interface Collection<E> extends Iterable<E> {  
    ...  
    boolean contains(Object o);  
    boolean add(E e);  
    boolean remove(Object o);  
    ...  
}
```

Методы:

1. `contains` — проверка на наличие элемента в коллекции.
2. `add` — добавление элемента в коллекцию (true — успешно, false — нет).
3. `remove` — удаление элемента из коллекции (true — если удалён).

COLLECTION

```
public interface Collection<E> extends Iterable<E> {  
    ...  
    boolean contains(Object o);  
    boolean add(E e);  
    boolean remove(Object o);  
    ...  
}
```

сравнивают объекты по `equals`

Q: Почему `contains` и `remove` принимают `Object`, а `add` — объект типа `E`?

A: Так решили разработчики: для `contains` и `remove` используется `equals`, а для `add` контролируется тип (чтобы мы не могли добавить объект другого типа).

COLLECTION

```
public interface Collection<E> extends Iterable<E> {  
    ...  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    ...  
}
```

сравнивают объекты внутри
коллекций по `equals`

Методы:

1. `containsAll` — проверка на вхождение другой коллекции (всех элементов).
2. `addAll` — добавление элементов другой коллекции в текущую (true — успешно, false — нет).
3. `removeAll` — удаление всех элементов, содержащихся в другой коллекции, из текущей (true — если удалён хотя бы один).

COLLECTION

```
public interface Collection<E> extends Iterable<E> {  
    ...  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean removeAll(Collection<?> c);  
    ...  
}
```

сравнивают объекты внутри
коллекций по `equals`

Q: А что это за вопросительные знаки?

A: Это wildcards, о которых мы с вами и должны поговорить.



WILDCARDS

```
public class ProductRepository {  
    private Collection<Product> items = new ArrayList<>();
```

```
    public Collection<Product> getAll() {  
        return items;  
    }
```

```
    public Product getById(int id) {  
        for (Product item : items) {  
            if (item.getId() == id) {  
                return item;  
            }  
        }  
        return null;  
    }
```

```
    public boolean add(Product item) {  
        return items.add(item);  
    }
```

```
    public boolean remove(Product item) {  
        return items.remove(item);  
    }
```

```
}
```

одна из реализаций*

Примечание*: у Collection нет прямой реализации

тип интерфейса, а не реализации

обратите внимание, насколько
реализация стала проще



IMPLEMENTATION

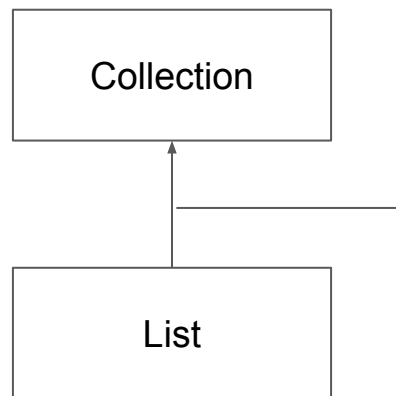
Интерфейс — это абстракция, мы можем указывать тип интерфейса в качестве типа переменной, параметра метода (и возвращаемого значения) или поля, но не можем написать `new Collection`.

Поэтому, нам нужна реализация — класс, который реализует интерфейс.

IMPLEMENTATION

У интерфейса `Collection` нет «прямой» реализации, поэтому, мы будем использовать `ArrayList`, реализацию интерфейса `List` (наследника `Collection`).

А раз `ArrayList` реализует интерфейс `List`, значит, он реализует и `Collection`.



наследование интерфейсов:
`List` «является» `Collection`

везде, где написано `Collection`, можно использовать `List`

Q & A

Q: Почему бы везде просто не написать ArrayList?

A: Тогда мы всех жёстко ограничим только одним типом. Суть интерфейсов состоит в максимальной гибкости; мы можем в любой момент только в одной точке поменять ArrayList на другую реализацию, и «внешний» код об этом не узнает.



Совет: старайтесь всегда* указывать **интерфейсы в качестве типа переменной, параметра метода (и возвращаемого значения) и типа поля.**

Примечание*: слово «всегда» — плохое (всегда исключения), но в большинстве случаев этот совет будет правильным.

REPOSITORY

Давайте добавим вспомогательные методы, которые позволяют:

1. Добавить целую коллекцию продуктов.
2. Удалить целую коллекцию продуктов.

```
public boolean addAll(Collection<Product> items) {  
    return this.items.addAll(items);  
}
```

```
public boolean removeAll(Collection<Product> items) {  
    return this.items.removeAll(items);  
}
```

REPOSITORY

```
class ProductRepositoryTest {  
    private ProductRepository repository = new ProductRepository();
```

```
@Test
```

```
void shouldAddProduct() {  
    repository.add(new Product());  
}
```

```
@Test
```

```
void shouldAddMultipleProducts() {  
    Collection<Product> products = new ArrayList<>();  
    products.add(new Product());  
    products.add(new Product());  
    repository.addAll(products);  
}
```

тип
интерфейса

конкретная
реализация



INHERITANCE

Но если мы попытаемся добавить коллекцию из объектов класса наследника, то ничего не скомпилируется (см. следующий слайд).

```
@Test
void shouldAddBook() {
    repository.add(new Book());
}
```

всё ок, Book является Product'ом

```
@Test
void shouldAddMultipleBooks() {
    Collection<Book> books = new ArrayList<>();
    books.add(new Book());
    books.add(new Book());

    repository.addAll(books);
}
```

Required type: Collection <Product> ⋮

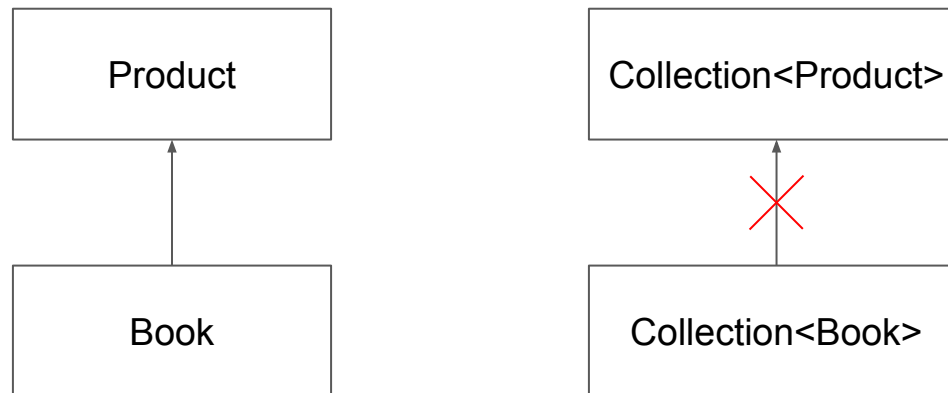
Provided: Collection <Book> не ок, Collection<Book> **не является** Collection<Product>

Change 1st parameter of method 'addAll' from 'Collection<Product>' to 'Collection<Book>'

Collection<Book> books = new ArrayList<Book>() ⋮

INHERITANCE

Важно запомнить: если между двумя типами есть отношения наследования, то при параметризации этими типами, таких отношений нет:



А значит, мы **не можем** использовать `Collection<Book>` там, где требуется `Collection<Product>`.

WILDCARD

Механизм, который позволяет нам использовать типы, параметризованные **другими типами**:

```
public interface Collection<E> extends Iterable<E> {  
    ...  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    ...  
}
```

любой тип,
начиная с Object

только E и его
наследники

Примечание*: помимо **extends** вы будете встречать ещё **super**, что будет означать только E и его родители (до Object).

WILDCARD

```
public boolean addAll(Collection<? extends Product> items) {  
    return this.items.addAll(items);  
}
```

```
public boolean removeAll(Collection<? extends Product> items) {  
    return this.items.removeAll(items);  
}
```

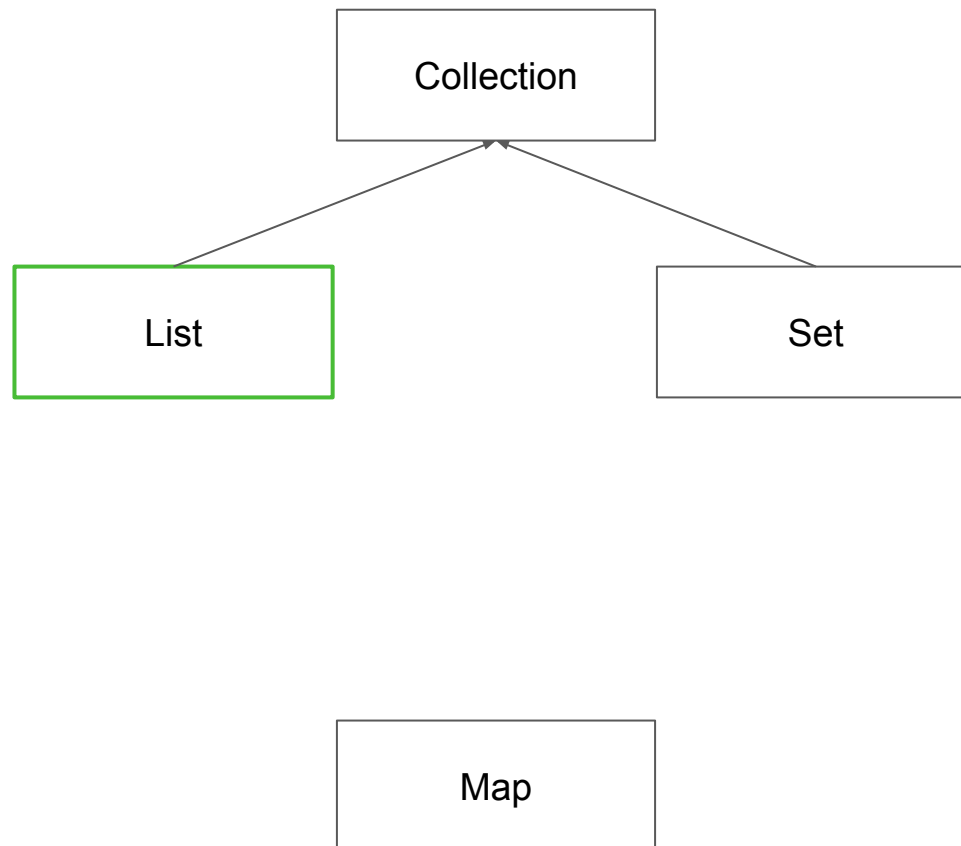
Q: Почему мы в методе удаления используем `extends`?

A: Мы говорим, что принимаем только `Product` и его наследников, чтобы пользователи нашего класса не передавали туда просто коллекцию объектов (хотя можно оставить и ?).



LIST

КЛЮЧЕВЫЕ ИНТЕРФЕЙСЫ



LIST

List — наследник Collection. Он определяет упорядоченную коллекцию элементов, в которой есть понятие индекса (т.е. порядкового номера элемента в коллекции):

```
public interface List<E> extends Collection<E> {  
    // все методы Collection +  
    void add(int index, E element);  
    E remove(int index);  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    E get(int index);  
    E set(int index, E element);  
    boolean addAll(int index, Collection<? extends E> c);  
    ...  
}
```

Примечание*: в большинстве случаев вы будете использовать именно List, поэтому обязательно познакомьтесь с набором его методов.

LIST

Большинство методов интуитивно понятны: поиск (`indexOf`, `lastIndexOf`) осуществляется по `equals`. Добавлены методы для работы по индексу.

```
public interface List<E> extends Collection<E> {  
    // все методы Collection +  
    void add(int index, E element);  
    E remove(int index);  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
    E get(int index);  
    E set(int index, E element);  
    boolean addAll(int index, Collection<? extends E> c);  
    ...  
}
```

```

public class ProductRepository {
    private List<Product> items = new ArrayList<>();

    public List<Product> getAll() {
        return items;
    }

    public Product getById(int id) {...}

    public boolean add(Product item) {...}

    public boolean remove(Product item) {...}

    public boolean addAll(Collection<? extends Product> items) {
        return this.items.addAll(items);
    }

    public boolean removeAll(Collection<? extends Product> items) {
        return this.items.removeAll(items);
    }
}

```

Q: Почему методы добавления и удаления остались с Collection?

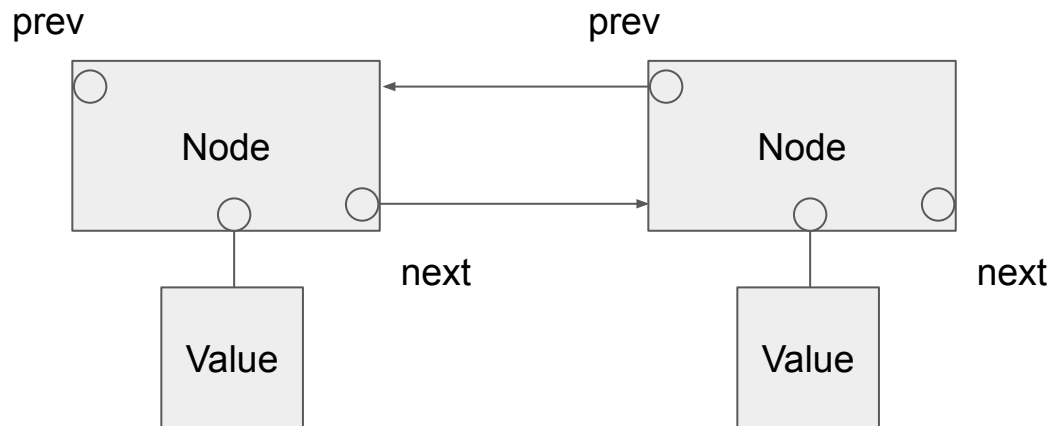
A: Нам важно обеспечить возможность добавления коллекции элементов (не обязательно списка).

РЕАЛИЗАЦИИ

У List есть две ключевых реализации: ArrayList и LinkedList.

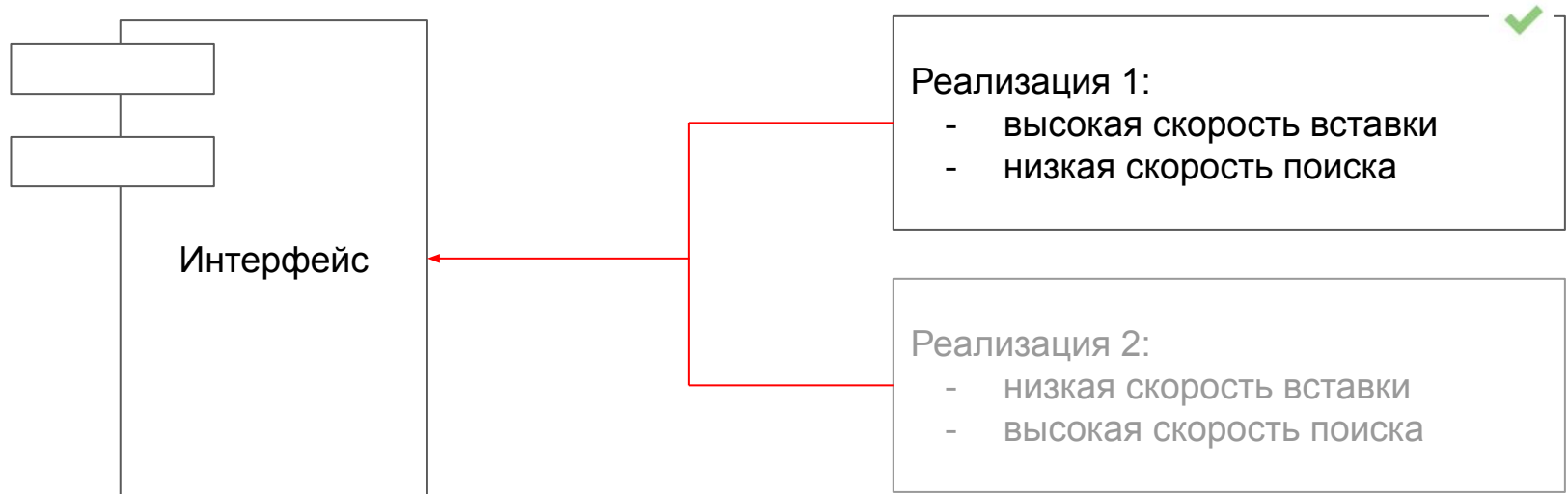
ArrayList основан на массивах (то, что мы раньше делали руками, только лучше).

LinkedList основан на СВЯЗАННЫХ СПИСКАХ:



РЕАЛИЗАЦИИ

На самом деле, для нас как для тестировщиков — это не очень важно, главное, что всё это организовано для возможности гибкой замены реализации при сохранении интерфейса:



РЕАЛИЗАЦИИ

Пример из жизни: у большинства лампочек более-менее стандартный цоколь (например, E27), при этом вы можете выбирать более яркую, тёплый/холодный свет и другие параметры, но систему электропроводки при этом менять не придётся.





DEFAULT

LIST

```
public interface List<E> extends Collection<E> {  
    ...  
    default void sort(Comparator<? super E> c) { ... }  
    ...  
}
```

Что такое **default**?



DEFAULT

Изначально интерфейсы были чистой абстракцией: в них можно было описывать только абстрактные методы (требования на реализацию).

Но с течением времени оказалось, что на базе уже описанных абстрактных методов можно написать и готовую реализацию, которую и помечают ключевым словом `default`.

Эта возможность появилась в Java 8.



DEFAULT

Q: Как это, «на базе уже описанных абстрактных методов»? Они же абстрактные — мы не можем их использовать, пока их кто-то не реализует?

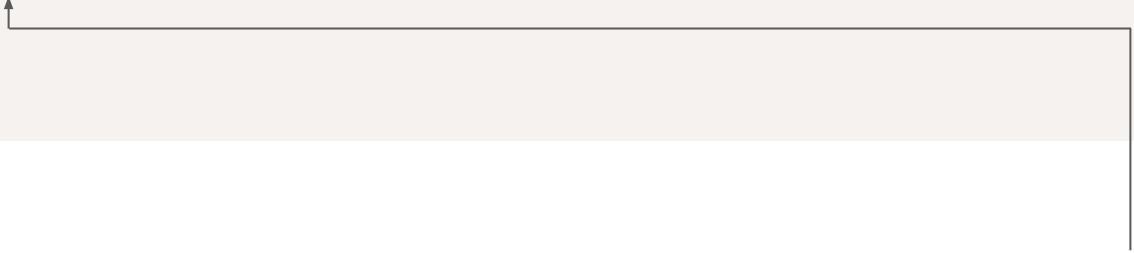
A: Мы можем это переформулировать так: в `default` методе можно вызывать любые другие методы, т.к. они всё равно будут реализованы в каком-то классе (который и реализует этот интерфейс).

Ключевое для нас: если метод помечен `default`, то необязательно его реализовывать в своём классе.

DEFAULT

В рамках ДЗ к предыдущей лекции рассматривается интерфейс `Comparator`, предназначенный для определения произвольного порядка (не натурального как в `Comparable`) объектов:

```
public interface Comparator<T> {  
    int compare(T o1, T o2); ← прямой порядок сортировки  
  
    default Comparator<T> reversed() { ← новый объект с обратным порядком сортировки  
        return Collections.reverseOrder(this);  
    }  
    ...  
}
```



Вопрос к аудитории: о чём вам говорит название класса `Collections`?



ADVANCED

COLLECTIONS

Достаточно часто вы будете видеть такое сочетание: интерфейс и утилитный класс к нему, в случае Collections Framework — это интерфейс `Collection` и класс `Collections` (обратите внимание: класса `Lists` нет, всё в `Collections`).

Напоминаем: утилитный класс — это вспомогательный класс, содержащий набор статических методов.

Начиная с Java 8, появилась возможность писать статические методы прямо в самих интерфейсах, поэтому, те связки (интерфейс + утилитный класс), которые сейчас есть, — останутся, но новые появляться не будет.

STATIC МЕТОДЫ В ИНТЕРФЕЙСАХ

```
public interface List<E> extends Collection<E> {  
    ...  
    static <E> List<E> of() {  
        return ImmutableList.emptyList();  
    }  
  
    static <E> List<E> of(E e1) {  
        return new ImmutableList.List12<>(e1);  
    }  
  
    static <E> List<E> of(E... elements) {  
        switch (elements.length) { // implicit null check of elements  
            case 0:  
                return ImmutableList.emptyList();  
            case 1:  
                return new ImmutableList.List12<>(elements[0]);  
            case 2:  
                return new ImmutableList.List12<>(elements[0], elements[1]);  
            default:  
                return new ImmutableList.ListN<>(elements);  
        }  
    }  
    ...  
}
```

набор «фабричных» методов
для быстрого создания списков

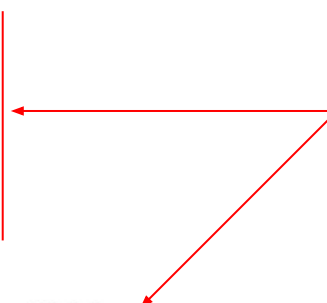
ФАБРИЧНЫЕ МЕТОДЫ

Фабричные методы — это способ удобного создания объектов необходимого типа:

@Test

```
void shouldAddMultipleProducts() {  
    Collection<Product> products = new ArrayList<>();  
    products.add(new Product());  
    products.add(new Product());  
    repository.addAll(products);  
  
    repository.addAll(List.of(new Product(), new Product()));  
}
```

4 строки
1 строка



GENERIC'И

Первое, что нужно разобрать, — generic в объявлении класса/интерфейса и generic в статическом методе:

```
public interface List<E> extends Collection<E> {  
    ...  
    static <E> List<E> of(E e1) {  
        return new ImmutableCollections.List12<>(e1);  
    }  
    ...  
}
```

Как бы странно это ни звучало, это разные :

1. Первое определяется, когда мы пишем `List<Product> = ...;`
2. Второе определяется, когда `List.of(new Product());`

GENERIC

Таким образом можно параметризовать отдельные методы:

```
public class ParametrizedType<T> {  
}
```


```
public class ParametrizedMethod {  
    public <T> void method(T param) { }  
    public static <T> void staticMethod(T param) { }  
}
```

это разные **T** — параметризуется уже не объект, а конкретный вызов метода

NESTED CLASSES & IMMUTABILITY

Интересна вот эта запись:

```
public interface List<E> extends Collection<E> {  
    ...  
    static <E> List<E> of(E e1) {  
        return new ImmutableCollections.List12<>(e1);  
    }  
    ...  
}
```



```
class ImmutableCollections {  
    static final class List12<E> extends AbstractImmutableList<E> implements Serializable {  
        ...  
    }  
}
```

NESTED

Внутри классов и интерфейсов можно объявлять классы и интерфейсы. Возможно это продвинутая тема для вашего уровня, вот ключевая идея:

1. Таким образом внутренний класс/интерфейс может получать доступ к `private` (либо `private static`, если объявлен как `static`) полям «объемлющего» класса
2. Таким образом не засоряется пространство имён пакета, т.к. вложенный интерфейс имеет полное имя:

`java.util.ImmutableCollections.List12`

имя пакета

IMMUTABLE

Q: Что значит immutable?

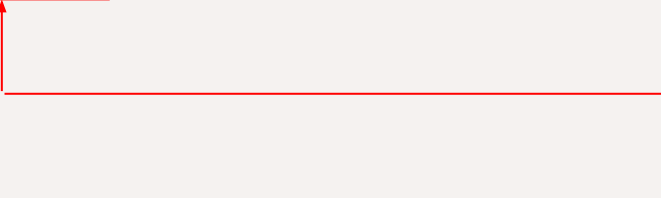
A: Это означает «неизменяемый» — при попытке любого изменения списка, созданного через `List.of`, мы получим исключение.

Q: Зачем так сделано?

A: Такие списки требуют меньше расхода памяти и защищены от случайного изменения. Подробнее вы можете прочитать [на странице официальной документации](#).

VARARG

```
public interface List<E> extends Collection<E> {  
    ...  
    static <E> List<E> of(E... elements) {  
        ...  
    }  
    ...  
}
```



Последнее, что нам осталось рассмотреть — это varargs.

Varargs — это специальная конструкция, когда после типа аргумента пишется «...». Это означает, что мы можем передавать в вызов этого метода произвольное количество аргументов, и все они будут аккуратно складываться в массив (в данном случае — `elements`).



CRUD

**ТЕСТИРОВАНИЕ СИСТЕМ,
УПРАВЛЯЮЩИХ НАБОРОМ
ОБЪЕКТОВ**



CRUD

CRUD (Create, Read, Update, Delete) — типовые операции, предоставляемые системами, которые управляют набором объектов.

Если мы посмотрим на системы, которыми пользуемся, то большинство из них будет предоставлять CRUD-операции.



МЕССЕНДЖЕРЫ

Мессенджеры предоставляют CRUD операции с сообщениями:

- создание
- редактирование
- удаление
- просмотр (списком с подгрузкой новых)
- поиск



СОЦСЕТИ

Соцсети предоставляют CRUD-операции почти со всеми существующими наборами объектов:

- посты
- комментарии
- лайки
- пользователи



СЕРВИСЫ ЗАКАЗА

Сервисы заказа (чего угодно: от книг и еды до авиабилетов) также предоставляют CRUD-операции:

- просмотр
- поиск
- заказ (фактически создание «брони»)

Стоит обратить внимание, что еще вводится разграничение прав доступа: вносить в систему некоторые виды объектов (например, рейсы) могут только лица, обладающие определёнными правами.

CRUD

Большую часть курса мы занимались созданием подобных CRUD-систем.

И методика их тестирования была примерно одинакова:

1. Проверяем систему в трёх состояниях:
 - a. Пустая система
 - b. В системе один объект
 - c. В системе больше одного объекта
2. Проверяем работу с существующими и несуществующими объектами (получение по существующему/несуществующему id, удаление по id)
3. Проверяем поиск на трёх возможных возвращаемых значениях:
 - a. Ничего не найдено
 - b. Найден ровно один объект
 - c. Найдено несколько объектов (тогда смотрим на их порядок)



CRUD

Ключевой вопрос: как быть с добавлением?

Ведь чтобы получить состояние системы с одним или несколькими элементами, в систему нужно эти элементы добавить.

Существующие варианты:

1. Вынести добавление в «преднастройку системы» (как мы и делали в некоторых ДЗ).
2. Сделать «вспомогательный» метод или конструктор, который позволяет создавать систему с нужным количеством объектов.



CRUD – ПРЕДНАСТРОЙКА СИСТЕМЫ

В случае с преднастройкой системы обычно success-поведение самого метода добавления не тестируют, т.к. если он будет работать некорректно, тесты, которые от него зависят «попадают».

При этом писать тесты на генерируемые исключения и бизнес-логику в самом методе добавления нужно.



CRUD – ВСПОМОГАТЕЛЬНЫЕ МЕТОДЫ

В случае со вспомогательными методами или конструкторами необходимо писать все тесты на метод добавления и следить за тем, чтобы вспомогательные методы и конструкторы не использовались «не тестами», либо покрывать тестами и их тоже.

Таким образом, как всегда — идеального решения не существует, вы должны выбирать и учитывать недостатки обоих способов.



ОРГАНИЗАЦИЯ КОДА

Здесь (как всегда) всё зависит от стиля принятого в определённой команде.

Два самых распространённых*:

1. В название класса теста добавляете название состояния:
EmptyProductRepositoryTest, SingleItemProductRepositoryTest (и другие вариации).
2. Внутри класса теста делаете вложенные классы, описывающие состояние, и уже в них тесты (см.следующий слайд).

Примечание*: конечно, есть и другие вариации (например, с классами, вложенными друг в друга матрёшкой)

ОРГАНИЗАЦИЯ КОДА

```
» public class CRUDRepositoryTest {  
    » @Nested  
    public class Empty {  
    }  
  
    » @Nested  
    public class SingleItem {  
    }  
  
    » @Nested  
    public class MultipleItems {  
    }  
}
```



SET



SET

Set — это коллекция, не содержащая повторяющихся элементов.

Повторяются элементы или нет, определяется через `equals`.

Коллекция не вводит понятия индекса, поэтому, методов для работы с индексами не предусмотрено.

Готовые реализации:

- *HashSet*
- *TreeSet*



MAP



MAP

Map — коллекция, содержащая пары ключ-значение, в которых ключи являются уникальными.

Уникальность ключей определяется через `equals`.

Важно: интерфейс Map не является наследником Collection.

Готовые реализации:

- *HashMap*
- *TreeMap*

MAP

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

тип для ключа (Key)

тип для значения (Value)

С этой коллекцией вы познакомитесь в рамках выполнения ДЗ к сегодняшней лекции.



ИТОГИ



ИТОГИ

Сегодня мы познакомились с Collections Framework и рассмотрели два ключевых интерфейса.

Вам обязательно нужно научиться им пользоваться (мы будем практиковать это на всём протяжении курса автоматизации).



ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

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



ИТОГИ КУРСА

Сегодня прошла наша итоговая лекция по курсу Java: он получился достаточно объёмным, но основная цель курса — дать вам инструмент и показать, как он используется (включая экосистему: Maven, JUnit и т.д.), именно с точки зрения потребностей автоматизатора.

Это значит, что вам обязательно нужно продолжать изучать Java и совершенствоваться (мы настоятельно рекомендуем книги Хорстмана).



КЛЮЧЕВОЕ

Важно: не попадайтесь в ловушку «сначала всё изучу, потом начну применять». Так не работает!

Как только получили новую информацию, познакомились с новым инструментом, — сразу применяйте его, в противном случае это будут «мёртвые» знания и навыки!



ЧТО ДАЛЬШЕ?

Впереди нас ждёт курс по автоматизации, на котором мы будем использовать навыки, которые приобрели на этом модуле, знакомиться с новыми инструментами и получать новые навыки (в том числе по Java).

Важно: на курсе по автоматизации мы будем считать, что вы обладаете всеми знаниями и навыками из программы «Java для тестировщиков». Поэтому, если что-то вам осталось непонятным, то обязательно напишите об этом в Slack.



Задавайте вопросы и напишите отзыв о лекции!

ВАСИЛИЙ ДОРОХИН

 Василий Дорохин