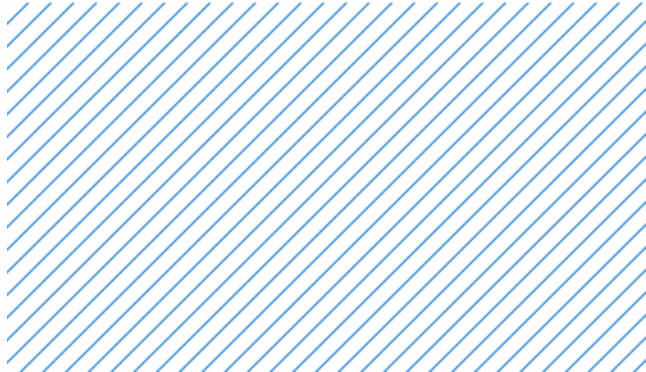
HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 11

Дженерики и коллекции

Программирование на языке Java

Роман Гуров
ВШЭ БИ 2021



Дженерики и наследование

Вспомним знакомое нам приведение к интерфейсу (или суперклассу):

```
Number number = 1;   // эквивалентно: = new Integer(1);
Number[] numberArray = new Integer[10];
```

Все ок, это отлично работает

Но, такое преобразование не сработает для типов дженерика:

```
Optional<Integer> optionalInt = Optional.of(1);
Optional<Number> optionalNumber = optionalInt;
```

Дженерики и наследование

```
Optional<Integer> optionalInt = Optional.of(1);
Optional<Number> optionalNumber = optionalInt;
```

Такое ограничение существует не случайно.
Допустим, что у `Optional` появился метод `.set()`, присваивающий ему некоторое значение:

```
public void set(T value) {
    if (value == null) {
        throw new NullPointerException();
    }
    this.value = value;
}
```

```
optionalInt.set(30);
```

Тогда, по логике вещей, после преобразования к `Optional<Number>`
станет законным вызов `set` от, например, `BigDecimal`

```
optionalNumber.set(new BigDecimal("3.14"));
```

Дженерики и наследование

Тот же самый пример, но с ArrayList:

```
ArrayList<Integer> arr_int = new ArrayList<>();
arr_int.add(42);
ArrayList<Number> arr_number = arr_int;
arr_number.add(new BigDecimal("3.14"));
```

Изначальный объект – массив интов, преобразование *ссылки на него* не должно изменять его сущность

Но стоп! А что же тогда тут?

```
Number[] numberArray = new Integer[10];
```

Само преобразование работает, но нарушить закон нам не позволит виртуальная машина:

```
numberArray[0] = 42;
numberArray[1] = 1.3; // ArrayStoreException
```

```
Exception in thread "main" java.lang.ArrayStoreException Create breakpoint : java.lang.Double
    at ru.hse.lecture11.Main.main(Main.java:24)
```

Дженерики и наследование

А зачем тогда вообще кому-то нужны такие преобразования?

```
Number[] numberArray = new Integer[10];
ArrayList<Number> arr_number = new ArrayList<Integer>();
```

Всё просто: как и раньше, мы хотим создавать универсальные функции

```
public static double doubleSum(Number[] values) {
    double sum = 0;

    for (int i = 1; i < values.length; ++i) {
        sum += values[i].doubleValue();
    }

    return sum;
}
```

Consumer и Supplier

В языке существует понятие функциональных интерфейсов:

```
public interface Consumer<T> {
    void accept(T value);
}
```

Аналог `void` функции,
принимающей один аргумент типа `T`

```
public interface Supplier<T> {
    T get();
}
```

Аналог функции без аргументов,
возвращающей объект типа `T`

Но самый фокус в том, что функции умеют приводиться к таким интерфейсам

```
Number number = 42;
Supplier<Integer> s = number::hashCode;
Consumer<Integer> c = System.out::println;
c.accept(s.get());
```

Подробности – в другой лекции

ifPresent и orElseGet в Optional

`Optional` совместим с этими функциональными интерфейсами:

```
Optional<String> optionalString = Math.random() > 0.5 ? Optional.of("test") : Optional.empty();
optionalString.ifPresent(System.out::println);
String result = optionalString.orElseGet(MyLocalizedMessages::GetEmptyStringMessageLocalized);
```

Попытаемся реализовать эти два метода:

```
class Optional<T> {
    private final T value;

    public void ifPresent(Consumer<T> consumer) {
        if (value != null) {
            consumer.accept(value);
        }
    }

    public T orElseGet(Supplier<T> supplier) {
        return value != null ? value : supplier.get();
    }
}
```

ifPresent и orElseGet в Optional

```
class Optional<T> {
    private final T value;

    public void ifPresent(Consumer<T> consumer) {
        if (value != null) {
            consumer.accept(value);
        }
    }

    public T orElseGet(Supplier<T> supplier) {
        return value != null ? value : supplier.get();
    }
}
```

Но в такой реализации всплывает проблема с первого слайда лекции:

```
Optional<CharSequence> opt_seq = /* ... */;
Consumer<Object> consumer = /* ... */;
Supplier<String> supplier = /* ... */;

opt_seq.ifPresent(consumer);
opt_seq.orElseGet(supplier);
```

Запрет на преобразование типов-параметров у дженерика не позволяет передавать вполне себе законные функциональные интерфейсы

Маски: `? super T` и `? extends T`

Вместо неконтролируемых преобразований дженериков
в языке используются маски (знаки вопроса вместо типа):

```
Consumer<Object> consumer = /* ... */;
Supplier<String> supplier = /* ... */;

Consumer<? super CharSequence> cons = consumer;
Supplier<? extends CharSequence> supp = supplier;
```

Компилятор разрешает преобразовывать к типу `SomeType<? super T>`
любые `SomeType`'ы с родителем `T` (и самим `T` тоже)

Аналогично, разрешает преобразовывать к типу `SomeType<? extends T>`
любые `SomeType`'ы с наследником `T` (и самим `T` тоже)

Маски: `? super T` и `? extends T`

Теперь, исправим методы:

```
class Optional<T> {
    private final T value;

    public void ifPresent(Consumer<? super T> consumer) {
        if (value != null) {
            consumer.accept(value);
        }
    }

    public T orElseGet(Supplier<? extends T> supplier) {
        return value != null ? value : supplier.get();
    }
}
```

Логично, что `Consumer`, принимающий родителя, нас устроит (потому что `value` спокойно к нему преобразуется)

И так же логично, что `Supplier` может вернуть наследника, ведь он легко преобразуется к `T`

Маска: одинокий ?

Что будет, если использовать ? без ограничений на тип?

```
Optional<?> optional = Optional.of(42);
```

Это эквивалентно `<? extends Object>`, то есть подходит любой тип

Но есть нюанс:

```
Optional<?> optional = Optional.of(42);
Object val1 = optional.get();
Object val2 = optional.orElse(40); // ошибка компиляции
```

Тот же нюанс на примере функциональных интерфейсов:

```
Consumer<?> consumer = System.out::println;
Supplier<?> supplier = Optional::empty;

consumer.accept(42); // ошибка компиляции
Object value = supplier.get();
```

Что происходит? Как это вообще объяснить??

Просто представьте, что вместо дженерика с маской может подставиться дженерик с любым удовлетворяющим маске типом

Тогда всё встаёт на свои места: 42 не подойдёт, например, в `Consumer<String>`, поэтому компилятор не позволяет вызывать метод

Коллекции

Коллекциями называют различные классы-контейнеры для однотипных элементов
Например, уже знакомый нам `ArrayList`

Почему нам не хватает просто массивов (например, `int[]`)?

- Их размер фиксирован, чтобы его изменить, придётся создать новый массив большего размера и перекопировать все элементы из старого

- Также, коллекция имеет много полезных методов:
от `equals` и `toString`, до операций пересечения и вычитания коллекций друг из друга

- Ещё, есть специальные версии коллекций, предназначенные для многопоточного использования, которые не ломаются при одновременном обращении из разных потоков исполнения

Классы коллекций – дженерики, поэтому коллекция *не может* хранить примитивный тип

Интерфейс `Collection`

Все коллекции реализуют интерфейс `java.util.Collection`:

```
package java.util;

public interface Collection<E> extends Iterable<E> {
    int size();

    boolean isEmpty();

    boolean contains(Object o);

    boolean add(E e);

    boolean remove(Object o);

    void clear();
}
```

Метод remove возвращает `boolean` – был ли найден удаляемый элемент в коллекции

Метод add делает то же самое наоборот, поскольку некоторые коллекции запрещают дубликаты

Для своей работы эти методы используют `equals`, поэтому важно, чтобы у хранимых объектов была правильная его реализация

и, как следствие, правильная реализация `hashCode()`

```
Collection<ComplexNumber> c = /* ... */;
c.add(new ComplexNumber(1, 2));
boolean contains = c.contains(new ComplexNumber(1, 2)); // ожидаем true
```

Итераторы и интерфейс `Iterable`

`Collection` наследует интерфейс `Iterable`, позволяющий получить **итератор**:

```
public interface Iterable<T> {
    Iterator<T> iterator();
    // ...
}
```

```
public interface Iterator<E> {
    boolean hasNext();

    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
}
```

Итератор позволяет *проитерироваться* по элементам коллекции

```
Collection<Integer> collection = new ArrayList<Integer>();
collection.add(5);
collection.add(7);

Iterator<Integer> it = collection.iterator();
while (it.hasNext()) {
    Integer element = it.next();
    System.out.println(element);
}
```

`hasNext()` говорит, есть ли очередной элемент

`next()` возвращает очередной элемент и сдвигает итератор на него

`remove()` удаляет из коллекции элемент, на который сейчас указывает итератор

Итераторы и интерфейс Iterable

```
Collection<Integer> collection = new ArrayList<Integer>();
collection.add(5);
collection.add(7);

Iterator<Integer> it = collection.iterator();
while (it.hasNext()) {
    Integer element = it.next();
    System.out.println(element);
}
```

Писать такой код каждый раз – не очень удобно, поэтому для итерирования есть другие способы:

Особая запись цикла for:

```
for (Integer element : collection) {
    System.out.println(element);
}
```

Работает с любым Iterable,
а также с простыми массивами

И метод коллекции forEach, принимающий Consumer

```
collection.forEach(System.out::println);
```

Интерфейс List

Одна из разновидностей коллекций – список – `List`:

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

    E get(int index);

    E set(int index, E element);

    void add(int index, E element);

    E remove(int index);

    int indexOf(Object o);

    int lastIndexOf(Object o);

    List<E> subList(int fromIndex, int toIndex);
}
```

`List` – простой список элементов, проиндексированных от `0` до `size() - 1`

`List` имеет методы, позволяющие работать с элементами по индексам:

- `set()` – замена элемента по индексу, возвращает старый элемент
- `remove()` – удаляет по индексу и возвращает удалённый элемент
- `add()` – ставит элемент в указанную позицию, сдвигая индексы всех мешающих элементов вперёд

Интерфейс List

`List` имеет методы, позволяющие работать с элементами по индексам:

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

    // ...

    int indexOf(Object o);

    int lastIndexOf(Object o);

    List<E> subList(int fromIndex, int toIndex);
}
```

- `indexOf()` – поиск элемента в массиве и возвращение индекса *первого* его вхождения
- `lastIndexOf()` – поиск элемента в массиве и возвращение индекса *последнего* его вхождения
- `subList()` – возвращает часть списка между двумя индексами (правая граница не включается)

Подсписок при этом не является копией, его изменения изменяют и оригинальный список:

```
List<String> words = /* ... */;

words.subList(1, 3).clear();
```

Реализации интерфейса List

Есть две основные реализации `List`'а: `ArrayList` и `LinkedList`

```
List<String> list1 = new ArrayList<>();
```

`ArrayList` – АТД "Динамический массив"

- Обращение по индексу – `O(1)`
- Вставка и удаление в конец – `O(1)` в среднем, иногда требуется расширение/сжатие с перекопированием
- Вставка/удаление в середине/начале – `O(N)`, требуется сдвиг всех элементы

```
List<Integer> list2 = new LinkedList<>();
```

`LinkedList` – АТД "Двусвязный список"

- Обращение по индексу – `O(N)`, надо дойти до элемента от начала списка
- Вставка и удаление в любое место – `O(1)` всегда, но не по индексу, а с помощью методов итератора листа

Также, среди всех коллекций принято иметь конструктор от `Collection`,
чтобы была возможность преобразования одной коллекции в другую

Вопрос

Какое слово нужно вписать вместо пропуска?

```
public static void appendHelloWorldToList(List<? {пропуск} String> list) {
    list.add("Hello World!");
}

public static void main(String []args) {
    List<String> list = new ArrayList<>();
    appendHelloWorldToList(list);
    list.forEach(System.out::println);
}
```

Вопрос

Какое слово нужно вписать вместо пропуска?

```
public static void appendHelloWorldToList(List<? super String> list) {
    list.add("Hello World!");
}

public static void main(String []args) {
    List<String> list = new ArrayList<>();
    appendHelloWorldToList(list);
    list.forEach(System.out::println);
}
```

Правильный ответ – `super`

Помним – на место вопроса встанет любой тип

Будь там наследник `String`, передать в `add` объект `String` было бы невозможно