**Функциональный стиль программирования** заключается в идее обращения с действиями как с объектами, которые можно сохранять в переменные и передавать в методы в качестве аргументов. Это помогает избежать изменения уже написанного кода класса и помогает избежать новых багов.

Например, пусть конструктор класса принимает объект интерфейса (как внешний аргумент функции), который реализует функционал преобразования строк. В этом случае, для переключения между реализациями не потребуется изменять уже написанный класс, а всего лишь передать нужный объект интерфейса.

*interface StringsTransformConfig {*

*String transform(String line); }*

*class StringsTransform {*

*private List<String> saved = new LinkedList<>();*

*private StringsSaverConfig config;*

*public StringsSaver(StringsTransformConfig config) {*

*this.config = config; }*

*public void save(String line) {*

*line = config.transform(line); }*

*public List<String> getSaved() return saved; }*

*class MyConfig implements StringsTransformConfig {*

*@Override*

*public String transform(String line) return "[" + LocalDateTime.now() + "] " + line; }*

**Анонимный класс** – специальный инструмент, позволяющий объявить одноразовый безымянный класс в момент создания его объекта. Для этого после объявления переменной и знака равно указывается родительский класс (или имплементируемый интерфейс), круглые скобки и фигурные скобки с реализацией. Внутри анонимных классов могут быть нестатические поля и методы. Все локальные переменные и параметры вне анонимного класса доступны ему только для чтения (они не должны потом изменяться в коде).



**Функциональный интерфейс** — это интерфейс, который содержит только один абстрактный метод (например, *Runnable*). Для его обозначения используется аннотация ***@FunctionalInterface***. Начиная с JDK 8, допускается чтобы функциональный интерфейс имел **методы по умолчанию**, которые не являются абстрактными, и статические методы.

Таким образом для реализации функционального стиля, объекты интерфейса можно передавать через сеттеры с анонимными классами, реализующими функциональные интерфейсы.

Функциональные интерфейсы могут быть обобщенными:

*interface SomeTest<T> {*

*boolean test(T n, T m); }*

*class GenericFunctionalInterfaceDemo {*

*public static void main(String args[]) {*

*SomeTest<Integer> isFactor = (n, d) -> (n % d) == 0;*

*if (isFactor.test(10, 2)) System.out.println(“2 является делителем 10”);*

*SomeTest<Double> isFactorD = (n, d) -> (n % d) == 0;*

*if (isFactorD.test(212.0, 4.0)) System.out.println(“4.0 является делителем 212.0”);*

*SomeTest<String> isIn = (a, b) -> a.indexOf(b) != -1;*

*String str = “Обобщенный функциональный интерфейс”;*

*if (isIn.test(str, “face”)) System.out.println(“’face’ найдено”); }*

Функциональные интерфейсы из стандартной библиотеки *java.util.function*:

* ***Consumer<T>*** содержит метод *accept(T)*, который ничего не возвращает. Удобен для вывода в консоль.
* ***Supplier<T>*** содержит метод *get()*, который возвращает объект типа *T*. Удобен для генератора чисел.
* ***Predicate<T>*** содержит метод *test(T)*, который возвращает булево значение. Удобен для проверок на *null*.
* ***Function<T, R>*** содержит метод *apply(T)*, который возвращает объект типа *R*. Удобен для парсинга числа из строки.
* ***UnaryOperator<T>*** содержит метод *apply(T)*, который возвращает объект типа *T*. Удобен для модернизации объектов.

**Лямбда-выражение** – специальный инструмент, позволяющий быстро и компактно имплементировать функциональный интерфейс. Оно появилось с *JDK* 8 и состоит из двух частей:

*(список входных параметров) -> { блок реализации функционального интерфейса }*

*() -> 98.6; // лямбда-выражения возвращает число*

*double myMeth() { return 98.6; } // обычный метод, возвращающий число*

Правила синтаксиса лямбда-выражений:

* У лямбды может быть 0 и более входных параметров.
* Параметры заключаются в круглые скобки и разделяются запятыми.

Например, *(String country, int size)*.

* Если Java может получить тип параметра из контекста, его можно не указывать.

Например, *(country, size)*.

* Если параметр один и его тип явно не указывается, скобки можно не писать.

Например, *distance -> return distance / 100*.

* Если параметров нет, нужно поставить пустые круглые скобки.

Например, *() -> "магазин закрыт"*.

* Тело лямбды может состоять из 0 и более выражений.
* Если тело состоит из одного оператора, его можно не заключать в фигурные скобки. Возвращаемое значение в этом случае можно указать без ключевого слова *return*.

Например, *distance -> distance / 100*.

* Во всех остальных случаях фигурные скобки обязательны, а в конце блока кода необходимо указать возвращаемое значение с использованием ключевого слова *return* (иначе будет возвращать *void*).

Сравним применение функционального интерфейса через анонимный класс и лямбда-выражение. Как видим, последнее выглядит более компактно и лаконично. Однако лямбды имеют более узкое применение, чем анонимные классы, т.к. не могут реализовывать несколько абстрактных методов и не поддерживают поля.

*public static void main(String[] args) {*

*Consumer<String> greeter = new Consumer<>() {*

*@Override // реализация через анонимный класс*

*public void accept(String name) {*

*System.out.println("Привет, " + name + "!"); } };*

*greeter.accept("Дарья"); }*

*public static void main(String[] args) { // реализация через лямбда-выражение*

*Consumer<String> greeter = name -> System.out.println("Привет, " + name + "!");*

*greeter.accept("Дарья"); }*

Лямбда-выражение имеет доступ к полям объекта, внутри которого она создана. Благодаря этому она реализует механизм **замыкания** – при изменении значения поля метод лямбды будет работать с новым значением этого поля. Лямбда также может менять эти поля.

Локальные переменные и параметры внутри лямбда-выражения называются **эффективно финальными**, т.е. неизменными.

Для модификации списка можно использовать метод **removeIf()**, который удаляет элементы, подошедшие под указанное условие. Рассмотри пример удаления пустых строк в списке. В таких случаях можно использовать **ссылки на методы**, позволяющие ссылаться на метод (статический и нестатический) без его выполнения.

*List<String> countries = new ArrayList<>(List.of("Мозамбик", "Япония", "", "США", ""));*

*countries.removeIf(name -> name.isEmpty()); // лямбда*

*countries.removeIf(String::isEmpty); // ИмяКласса::метод*

**Ссылка на метод экземпляра**. Проверим содержится ли указанный объект в списке.

*Predicate<String> hasCountry = str -> countries.contains(str); // лямбда*

*Predicate<String> hasCountry = countries::contains; // объект::метод*

**Ссылка на конструктор**. Метод *apply* функционального интерфейса *Function* принимает число и возвращает *ArrayList* (отсылка к конструктору с вместимостью списка).

*Function<Integer, ArrayList<Integer>> listCreator = capacity -> new ArrayList<>(capacity); // лямбда*

*Function<Integer, ArrayList<Integer>> listCreator = ArrayList::new; // Класс::new*

Лямбда-выражения могут генерировать исключения совместимые с исключениями реализуемого абстрактного метода.

Дженерик-интерфейс ***Stream*** – встроенный инструмент *java* для потоковой обработки данных в функциональном стиле. Стрим можно создать через статические методы ***Stream.of()***, ***Stream.iterate()*** либо через метод коллекции ***stream()***. Также можно получить стрим из строк файла через ***Files.lines(Path path)***.

*Stream<String> names = Stream.of("Арсений", "Иван", "Аня");*

*List<String> citiesToVisit = List.of("Лондон", "Мадрид", "Владивосток");*

*Stream<String> visitStream = citiesToVisit.stream();*

Операции над стримами бывают двух видов – промежуточные и терминальные.

**Терминальные** – конечная операция, вызывающаяся после промежуточных. Вызывается один раз.

* Операция ***forEach()*** применяет переданный в нее параметр ко всем элементам стрима. Например, можно вывести в консоль все, что передано в стрим.

*Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.forEach(System.out::println);*

* Операция ***count()*** считает число элементов стрима (после всех фильтров).

*long count = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.count();*

* Операция ***allMatch()*** проверяет, что все ли элементы стрима удовлетворяют условию-параметру.

*boolean isAllFilm = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.allMatch(title -> title.startsWith("фильм"));*

* Операция ***anyMatch()*** проверяет, что хотя бы один элемент удовлетворяют условию-параметру.

*boolean found = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.anyMatch(title -> title.toLowerCase().contains("спринг"));*

* Операция ***noneMatch()*** проверяет, что все элементы не удовлетворяют условию-параметру.

*boolean found = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.noneMatch(title -> title.toLowerCase().contains("тест"));*

* Операция ***findFirst()*** возвращает первый элемент стрима.

*Optional<String> maybeFound = Stream.of("сериал «Спринг»", "сериал «Алгоритмы»", "фильм «Матрица»")*

*.findFirst();*

* Операция ***findAny()*** возвращает любой элемент стрима.

*Optional<String> maybeFound = Stream.of("сериал «Спринг»", "сериал «Алгоритмы»", "фильм «Матрица»")*

*.findFirst();*

* Операция ***collect()*** позволяет собрать элементы стрима в коллекцию или другой объект.

*List<String> collected = Stream.of("сериал «Спринг»", "сериал «Алгоритмы»", "фильм «Матрица")*

*.collect(Collectors.toList());*

* Операции ***min()*** и ***max()*** позволяют найти минимальный и максимальный элементы стрима.
* Операция ***sum()*** позволяют найти сумму элементов стрима.
* Операция ***average()*** позволяют найти среднее значение элементов стрима.
* Операция ***summaryStatistics()*** считает всю статистику стрима (*count, sum, min, max, average*).
* Операция ***reduce()*** позволяет собрать элементы стрима в один поток (обратный аналог *flatMap()*).

*int reduceResult = Stream.iterate(1, i -> i <= 5, i -> i + 1)*

*.reduce(1, (acum, cur) -> acum \* cur); // рассчет факториала от 1 до 5*

**Промежуточные** – задают правила изменения стрима, возвращая трансформированный поток исходных данных. Например, фильтрация. Может быть задано несколько промежуточных операций.

* Операция ***filter()*** возвращает новый стрим из элементов удовлетворяющих заданному предикату.

*Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.filter(title -> title.contains(“фильм”)*

*.forEach(System.out::println);*

* Операция ***map()*** применяет лямбду к элементам стрима и возвращает новый стрим. Каждому элементу исходного стрима соответствует один элемент нового стрима.

*Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.map(ratings::get) // значение берется из хэшмапы ratings*

*.forEach(System.out::println);*

* Операция ***flatMap()*** применяет лямбду к элементам стрима и возвращает новый стрим. Каждому элементу исходного стрима соответствует 0, 1 или несколько элементов нового стрима.

*Stream.of("ПРОРЫВ В ЭКОНОМИКЕ", "ЛУЧШЕЕ ДЕТЯМ", "ЛУЧШЕЕ В ЭКОНОМИКЕ")*

*.flatMap(sentence -> Arrays.stream(sentence.split(" ")))*

*.forEach(System.out::println);*

* Операция ***limit()*** ограничивает стрим указанным количеством элементов.

*Stream.of("Пирамиды", "Эверест", "Колизей")*

*.limit(2)*

*.forEach(System.out::println);*

* Операция ***skip()*** отбрасывает указанное количество первых элементов стрима.

*Stream.of("Пирамиды", "Эверест", "Колизей")*

*.skip(2)*

*.forEach(System.out::println);*

* Операция ***distinct()*** позволяет получить стрим из уникальных элементов исходного стрима.

*Stream<Integer> names = Stream.of("Надя", "Ольга", "Марат", "Света", "Ева")*

*.map(String::length)*

*.distinct();*

*names.forEach(System.out::println);*

* Операция ***peek()*** позволяет выполнить дополнительное действие над элементами стрима без влияния на тип стрима.

*Stream.of(1, 1, 2, 2, 3, 3)*

*.peek(i -> System.out.println(“peek: “ + i)*

*.distinct()*

*.forEach(System.out::println);*

* Операция ***sorted()*** накапливает в себе элементы стрима, сортирует их и отправляет дальше.

*new Random().ints()*

*.limit(10)*

*.sorted()*

*.forEach(System.out::println);*

* Операции ***takeWhile()*** и ***dropWhile()*** с условиями (аналог циклов while). В момент, когда получает первый *false*, перестает работать (выходит из цикла) и пропускает все остальные элементы.

*Stream.of(1, 1, 2, 2, 3, 3)*

*.takeWhile(i -> i < 2)*

*.forEach(System.out::println);*

* Операция ***mapToObj()*** позволяет выполнить преобразование элемента стрима и запаковать его в объект.

*String str = “Hello world!”;*

*str.chars()*

*.mapToObj(c -> (char) c)*

*.forEach(System.out::println);*

Промежуточные операции выполняются только при вызове терминальной операции. Такое действие называется **ленивостью** (***laziness***).

В стримах также есть аналог цикла *for*:

*Stream.iterate(0, v -> v < 10, v -> v + 1) // Вывод чисел от 0 до 9*

*.forEach(System.out::println);*

У структур данных (коллекции, списки, массивы и т.д.) есть свой метод преобразования в стрим.

Для проверки защиты от исключения *NullPointerException* можно использовать класс-обертку ***Optional<T>***. У него есть следующие методы:

* Метод ***isPresent()*** возвращает *true*, если внутри не *null*.
* Метод ***ifPresent()*** позволяет выполнить какое-либо действие, если объект не пустой.
* Метод ***get()*** возвращает распакованное значение.
* Метод ***orElse(value)*** в случае отсутствия внутри значения вернет указанный параметр.
* Метод ***orElseThrow()*** в случае отсутствия значения выбросит исключение.
* Метод ***Optional.of(value)*** создает класс-обертку с указанным параметром.
* Метод ***Optional.empty()*** создает пустой класс-обертку.

Для примитивных типов также существуют классы-обертки. Например, ***OptionalInt*** и ***OptionalDouble***. Для некоторых примитивов есть собственные типизированные стримы – ***IntStream.range(int, int)***.

Подробнее можно прочитать [тут](https://habr.com/ru/post/346782/).

Класс-обертка ***Optional*** позволяет удобно обработать возможные null-значения.

*@Override*

*public Optional<User> findUserById(String id) {*

*SqlRowSet userRows = jdbcTemplate.queryForRowSet("select \* from cat\_user where id = ?", id);*

*if(userRows.next()) {*

*User user = new User( userRows.getString("id"),*

*userRows.getString("username"),*

*userRows.getString("nickname") );*

*return Optional.of(user);*

*} else {*

*log.info("Пользователь с идентификатором {} не найден.", id);*

*return Optional.empty(); } }*