Колбеки и цепь обязанностей

Введение в лямбда выражения

Содержание

1. Интерфейс колбека
2. Цепочка обязанностей
3. Лямбда выражения

1. Интерфейс колбека

Итак, начнем нашу увлекательную лекцию с задачи которую вы могли бы решать в прошлой лекции. Сразу скажу, мое решение может показаться сложным, потому что будет иметь в себе не только явное использование принципов ООП но и использование 1 патерна проектирования. Итак, приступим

Мы попробуем имитировать работу целой ИТ команды. Конечно же с некими упрощениями.

Предположим что команда состоит из Дизайнера, Программиста и Тестировщика. Для упрощения мы предположили что задачи уже придуманы верхнеуровнего продактоунером. Теперь, давайте начнем с простого, а именно опишем класс задачи.

Итак, из чего же состоит класс Задачи? Во-первых нам нужен айдишник, уникальное число для того чтобы различать задачи. Далее нам нужен статус задачи. Для этого давайте напишем простой енам из состояний – требования собраны, готово для работы, в прогрессе, готово для тестирования и завершено. Обьясню для тех, кто не очень понимает или никогда не имел дел с реальными задачами.

Сначала некий человек (продактоунер) придумывает задачу, типа – нужно сделать возможность смотреть профиль юзера. Далее придумываются требования к задаче аналитиком, но эту часть мы опустим в нашей задаче. Далее Дизайнер на основе требований рисует дизайн. В нашей задаче я поставил в ответственность дизайнеру еще и тесткейсы написать, но это не так в реальной жизни и это только для упрощения. Далее, когда дизайнер сделал свою работу он добавляет ссылку на дизайн и ссылку на тесткейсы внутрь задачи. И ставит статус задачи – готово для выполнения. В этот момент задачу уже может брать Программист и делать ее. Когда он ее сделает, то итогом он добавит в задачу ссылку на билд (сборку) и поставит статус Готово к тестированию. Программист не может делать задачу без дизайна и тесткейсов и потому мы требуем их перед тем как брать задачу в работу. Далее Тестировщик берет задачу и смотрит на тесткейсы и по ссылке на сборку ставит ее и тестирует. Если все ОК то задача переходит в статус DONE. Это конечно же упрощение, потому что часто после тестирования возникает необходимость доработать задачу.

Итак, у задачи есть айди, статус, описание, ссылка на дизайн, ссылка на тесткейсы и ссылка на сборку. Понятное дело, что в процессе работы этих данных нет и они создаются по пути. Но мы не будем давать возможность менять обьект класса и потому сделали все поля класса

private final. Да, мы сделали паблик геттеры для работы с полями, но на данном этапе это лишь упрощение.

```
public class Task {

    private final int id;
    private final Status status;
    private final String description;
    private final String designLink;
    private final String testcase;
    private final String buildLink;

    public Task(int id,
                Status status,
                String description,
                String designLink,
                String testcase, String buildLink) {...}

    public int getId() { return id; }

    public Status getStatus() { return status; }

    public String getDescription() { return description; }

    public String getDesignLink() { return designLink; }

    public String getTestcase() { return testcase; }

    public String getBuildLink() { return buildLink; }

    enum Status {
        ASSEMBLING_REQUIREMENTS,
        READY_TO_DO,
        IN_PROGRESS,
        READY_FOR_TESTING,
        DONE
    }
}
```

Вместо того, чтобы у обьекта менять его поля, мы заменим старый обьект на новый когда один из работников завершит свою работу. Ладно. Давайте тогда напишем класс для работника. Конечно же он будет абстрактным, ведь разные работники будут иметь свои особенности.

Итак, наш абстрактный работник должен иметь имя, а также у него должен быть статус задачи, по которому он будет брать задачу. Ведь согласитесь, разработчик не может брать задачу в работу, если она еще на стадии получения требований и нет дизайна. Так же разработчик не сможет брать задачу если он ее уже сделал и отправил на тестирование.

Если интересно, погуглите доски канбан, это несколько столбцов где задачи идут слева направо. И у каждого столбца есть название – TO DO, IN_PROGRESS, TESTING, DONE.

```java
public abstract class Employee {

    private final TaskProgressCallback callback;
    private final String name;
    private final Task.Status taskStatus;

    protected Employee(TaskProgressCallback callback,
                       String name,
                       Task.Status taskStatus) {
        this.callback = callback;
        this.name = name;
        this.taskStatus = taskStatus;
    }

    public void doTask(Task task) {
        System.out.println(getClass().getSimpleName() + " " + name
                + " is doing task " + getDetails(task));
        callback.updateTask(getTaskWhenDone(task));
    }

    public Task.Status getTaskStatus() {
        return taskStatus;
    }

    protected abstract Task getTaskWhenDone(Task task);

    protected abstract String getDetails(Task task);
}
```

И что это за поле первое? Колбек прогресса. Нам нужно уведомить всех других когда мы завершим работу. Давайте посмотрим на метод doTask. Мы получаем задачу, отписываемся в консоль что работник такого типа с таким именем начал делать задачу и какие-то детали. Когда мы завершим работу, то вызовем у интерфейса его метод, в который передадим новое состояние задачи. Поймем на первом примере, не переживайте. Также нам нужно отдать наружу методом гет какой статус задачи может брать работник.

```java
 */
public interface TaskProgressCallback {

    void updateTask(Task task);
}
```

Да, как видите интерфейс с 1 методом, ничего необычного. Когда работник завершил задачу то он все свои наработки добавляет в новую задачу и отправляет обратно на доску канбан.

Кстати, вы заметили что я положил енам внутрь класса задачи? Это очень удобно, потому что состояние задачи имеет лишь отношение к задаче. Т.е. я бы мог написать новый енам класс в отдельном файле, но вместо этого написал в одном файле. И сразу понятно что статус относится к задаче.

```java
public class Designer extends Employee {

    protected Designer(TaskProgressCallback callback, String name) {
        super(callback, name, Task.Status.ASSEMBLING_REQUIREMENTS);
    }

    @Override
    protected Task getTaskWhenDone(Task task) {
        return new Task(
                task.getId(),
                Task.Status.READY_TO_DO,
                task.getDescription(),
                getDesignLinkForTask(task.getId()),
                getTestCaseForTask(task.getDescription()),
                buildLink: ""
        );
    }

    @Override
    protected String getDetails(Task task) {
        return " with taskId" + task.getId() + " and description " + task.getDescription();
    }

    private String getDesignLinkForTask(int taskId) {
        return "https://project/design_link_for_task_with_id " + taskId;
    }

    private String getTestCaseForTask(String taskDescription) {
        return "when " + taskDescription + " get result " + Math.random();
    }
}
```

Ладно, рассмотрим конкретный вид работника Дизайнер. Во-первых мы передаем в конструктор родителя статус задачи ASSEMBLING_REQUIREMENTS потому что дизайнер собственно дополняет задачу дизайном и тесткейсами (в реальности нет, мы упростили здесь). Далее посмотрим на метод, который отдает новую задачу когда она готова – мы создаем новую задачу и берем айди который был до работы, статус меняем на READY_TO_DO чтобы следующий работник смог взять задачу в работу, описание не меняется, но мы генеирируем линк для дизайна и для тесткейсов и в методе детальной информации мы пишем с чем имели дело – задача с айди и описанием. Ведь больше пока ничего не было у дизайнера. Чтобы понять еще лучше, посмотрим на класс Программиста.

Итак, программист это тот работник, который берет задачи, которые готовы к работе т.е. READY_TO_DO. Сначала дизайнер взял задачу в статусе подготовки требований и когда закончил поменял статус на готово к работе, здесь ее перехватит разработчик/программист и будет ее делать. Когда он закончит свою работу, то возьмет данные задачи и добавит к ним ссылку на сборку. Также поменяет ей статус на READY_FOR_TESTING, т.е. готово к тестированию. Когда разработчик будет работать над задачей он отпишется что работает с задачей у которой айди такое и ссылка на дизайн и тесткейсы таковые.

Ладно, вроде понятно. Берем задачу, все что было до нас готово оставляем как было, а то что мы добавили пишем в новый обьект задачи и отправляем колбеку на обработку. Нам неважно и работники не знают что и как будет дальше с задачей, для этого и есть колбек интерфейс. Мы не передаем работнику фабрику задач, а общаемся с помощью простого интерфейса. Сделал дело - отдал интерфейсу на обработку и все. Дальше сами разберутся с этой задачей.

```java
public class Programmer extends Employee {

    protected Programmer(TaskProgressCallback callback, String name) {
        super(callback, name, Task.Status.READY_TO_DO);
    }

    @Override
    protected Task getTaskWhenDone(Task task) {
        return new Task(
                task.getId(),
                Task.Status.READY_FOR_TESTING,
                task.getDescription(),
                task.getDesignLink(),
                task.getTestcase(),
                getBuildLinkForTask(task.getId())
        );
    }

    @Override
    protected String getDetails(Task task) {
        return " with task id" + task.getId() +
                " and designLink" + task.getDesignLink() +
                "\nand testcase " + task.getTestcase();
    }

    private String getBuildLinkForTask(int taskId) {
        return "https://bitbucket/project/link_for_build_with_task_id_" + taskId;
    }
}
```

```java
public class Tester extends Employee {

    protected Tester(TaskProgressCallback callback, String name) {
        super(callback, name, Task.Status.READY_FOR_TESTING);
    }

    @Override
    protected Task getTaskWhenDone(Task task) {
        return new Task(
                task.getId(),
                Task.Status.DONE,
                task.getDescription(),
                task.getDesignLink(),
                task.getTestcase(),
                task.getBuildLink()
        );
    }

    @Override
    protected String getDetails(Task task) {
        return "with id " + task.getId() + "and testcase " + task.getTestcase();
    }
}
```

Заметьте, программист когда закончил работу с задачей поменял ей статус на READY_FOR_TESTING. А тестировщик это тот человек, который может брать задачи только

со статусом READY_FOR_TESTING. Когда же тестировщик заканчивает работу он просто меняет ей статус на DONE и все.

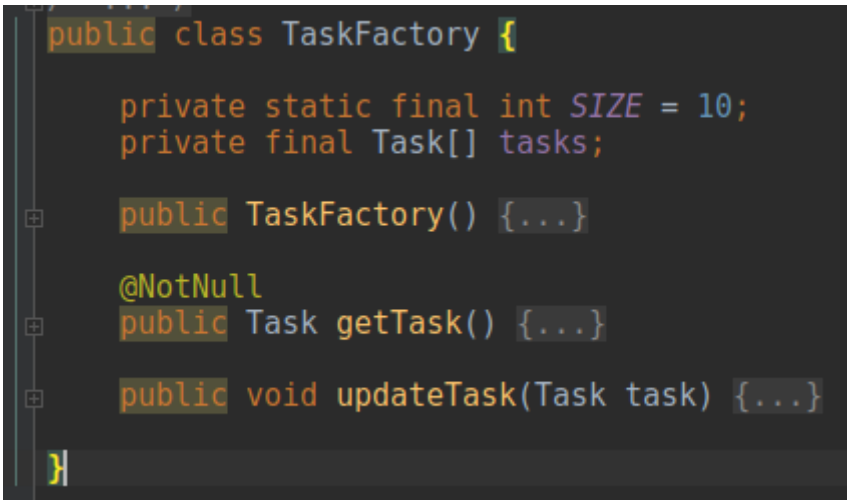
2. Цепочка обязанностей

Ладно, мы написали класс для задачи, написали класс для работника и 3 реализации. Что дальше? Как нам связать их всех? Чтобы была цепочка из работников? Немного теории – есть такой патерн (о них всех мы поговорим подробнее позже) который называется цепочка обязанностей. Суть в том, что мы связываем работников и если один может обработать задачу, то берет и делает ее, но если у задачи не тот статус, то он передает ее следующему по цепочке. Давайте посмотрим на класс ЦепочкаРаботника и все будет ясно.

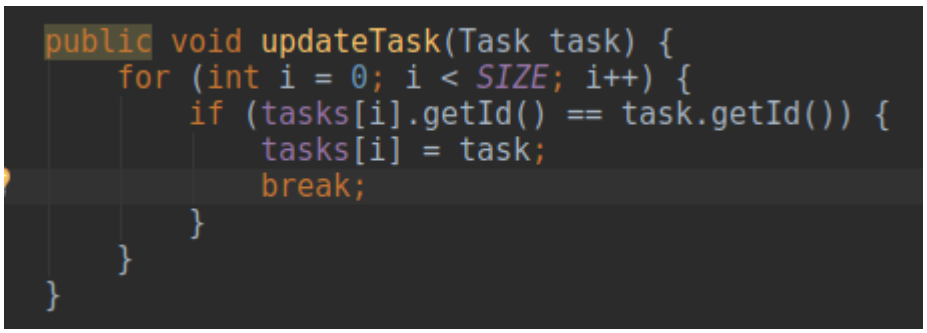
```
public class EmployeeChain {

    private final Employee employee;
    private EmployeeChain nextEmployeeChain;

    public EmployeeChain(Employee employee) {
        this.employee = employee;
    }

    public void doTask(Task task) {
        if (task.getStatus() == employee.getTaskStatus()) {
            employee.doTask(task);
        } else if (nextEmployeeChain != null) {
            nextEmployeeChain.doTask(task);
        } else {
            throw new IllegalArgumentException("task can't be handled");
        }
    }

    public void setNextEmployee(EmployeeChain nextEmployeeChain) {
        this.nextEmployeeChain = nextEmployeeChain;
    }
}
```

Наш класс обертка над работником принимает в аргумент конструктора работника. Далее у него есть метод который добавляет в цепочку другого работника (мы не требуем в конструктор просто потому что не хотим передавать нул, можно иначе, я позже покажу как когда будем проходить цепочку детально). Теперь, метод сделатьЗадачу (doTask) простой – смотрим на статус задачи, если он совпадает со статусом задачи работника, то он ее делает, если же нет, то смотрим на второго работника в цепочке, если он есть, то отдаем ему на обработку эту задачу. Если же нет второй цепочки, то можем бросить исключение типа – задача не может быть обработана. Сразу обьясню зачем так – когда задача перейдет от дизайнера к программисту, а потом от него к тестировщику, то после тестирования никто уже не будет ничего с ней делать. Поэтому и с помощью выброса исключения мы поймем что задача завершена. Можно сделать иначе конечно, например засетить классу интерфейс что задача не была обработана работником и пусть там происходит все что угодно. Мы в дальнейшем сможем переписать этот код.

Ладно, мы связали работников в цепочку, но так же нам нужна фабрика задач. Напишем же ее.



Здесь я нажал на Ctrl+Shift+”-” и все методы схлопнулись. Можете нажать на Ctrl+Shift+”+” и увидеть весь код. Теперь, давайте у нас будет массив из задач. И 2 метода – получить задачу и обновить ее. Метод обновления задачи довольно-таки простой.



Находим в массиве задачу, у которой айди совпадает с новой задачей и перезаписываем ее полностью. И посмотрим на конструктор тоже

Мы просто создаем массив и заполняем его в цикле. Обратите вминание на статус и что кроме айди и описания ничего больше нет.

Ладно, что же будет в методе получения задачи?

Мы не можем брать задачи у которых статус уже DONE значит найдем первую же задачу у которой статус будет иной и все. Если же мы дойдем до момента, когда не останется таких задач то просто вернем первую задачу в массиве. Если помните, наша цепочка работников может обработать все задачи кроме той, у которой статус DONE. Хотя это просто решить –

напишем заглушечный работник типа бот и он просто отпишется в консоль что завершили задачу. Давайте пока так посмотрим на ситуацию и потом уже поменяем.

```
@NotNull
public Task getTask() {
    Task result = null;
    for(Task task : tasks) {
        if (task.getStatus() != Task.Status.DONE) {
            result = task;
            break;
        }
    }

    if (result == null) {
        result = tasks[0];
    }
    return result;
}
```

Ладно. Разобрались пока что со всем этим. Теперь осталось самое сложное, связать все эти классы в одну программу.

3. Лямбда выражения

```
public static void main(String[] args) {
    TaskFactory factory = new TaskFactory();
    TaskProgressCallback callback = new TaskProgressCallback() {
        @Override
        public void updateTask(Task task) {
            factory.updateTask(task);
        }
    };
    EmployeeChain chain = new EmployeeChain(new Designer(callback, name: "Alycia"));
    EmployeeChain next = new EmployeeChain(new Programmer(callback, name: "John"));
    EmployeeChain last = new EmployeeChain(new Tester(callback, name: "Steve"));
    next.setNextEmployee(last);
    chain.setNextEmployee(next);

    while (true)
        chain.doTask(factory.getTask());
}
```

Давайте пройдем по всем линиям и поймем что же здесь происходит.

Итак, первая линия в мейн методе – создаем фабрику задач. Все просто. Вторая линия – нам нужен колбек, но мы знаем что это интерфейс и интерфейсы нужно имплементировать в классах, но секунду, мы же забыли это сделать. Что же у нас здесь происходит? Я написал что интерфейс равно новый интерфейс? ЧАВО? Так можно было? Да, это называется анонимный класс. Если вам лень писать класс вы можете поступить так. Как видите идеа подчеркнула серым наш код new TaskProgressCallback и угадайте что можно сделать? Нет, пока что мы этого делать не будем. Шучу. Давайте. Нажмите на Alt+Enter.

```
TaskFactory factory = new TaskFactory();
TaskProgressCallback callback = task -> factory.updateTask(task);
EmployeeChain chain = new EmployeeChain(new Designer(callback, name: "
```

Вау. Что за стрелка? Это называется лябмда выражение. Но секунду, идеа подчеркула желтым еще раз, а ну-ка.

```
TaskFactory factory = new TaskFactory();
TaskProgressCallback callback = factory::updateTask;
EmployeeChain chain = new EmployeeChain(new Designer(c
```

С ума сойти. Что это значит? Это значит что мы вызовем метод у фабрики который называется updateTask. И оно все там само разрулит. Ладно, если вам сложно это читать, то вернемся к нашему певоначальному виду. Как мы могли бы написать класс вместо анонимного? А просто написать класс у которого в конструктор подается фабрика и там вызывается метод.

```
public class CallbackImpl implements TaskProgressCallback {

    private final TaskFactory taskFactory;

    public CallbackImpl(TaskFactory taskFactory) {
        this.taskFactory = taskFactory;
    }

    @Override
    public void updateTask(Task task) {
        taskFactory.updateTask(task);
    }
}
```

```
public static void main(String[] args) {
    TaskFactory factory = new TaskFactory();
    TaskProgressCallback callback = new CallbackImpl(factory);
    EmployeeChain chain = new EmployeeChain(new Designer(callback, name: "Alycia"));
    EmployeeChain next = new EmployeeChain(new Programmer(callback, name: "John"));
    EmployeeChain last = new EmployeeChain(new Tester(callback, name: "Steve"));
    next.setNextEmployee(last);
    chain.setNextEmployee(next);

    while (true)
        chain.doTask(factory.getTask());
}
```

Итак, мы создали колбек и инициализировали классом, который принимает фабрику и вызывает метод обновления задачи. Хорошо. Далее у нас идет цепочка работников.

Создаем цепь. В конструктор отдаем дизайнера и его имя. Создаем вторую цепь next и там отдаем Программиста по имени John. Последним звеном будет Тестировщик. Всем работникам отдаем колбек, чтобы они обновляли задачу в фабрике когда завершат работать с ней. Но мы еще должны связать цепочку. Для этого второму звену добавляем последнее звено

как следующее и первому звену сетим второе звено как последнее. Надеюсь все понятно. Дизайнер → Программист → Тестировщик.

И самое последнее – в бесконечном цикле вызываем метод у цепи и передаем задачу из фабрики. Мы не прерываем цикл потому что знаем, что его прервет выброс исключения в цепочке. Ведь что произойдет? Сначала задачи пойдут выполняться по цепи и когда все выполнятся наш метод получения задачи из фабрики вернет первую со статусом DONE. Дизайнер не обработает и отдаст программисту, а тот в свою очередь отдаст тестиворщику и за неимением следующей цепи выбросится исключение.

Давайте же запуситм код!

```
Designer Alycia is doing task  with taskId0 and description description 0
Programmer John is doing task  with task id0 and designLinkhttps://project/design_link_for_task_with_id 0
and testcase when description 0 get result 0.8571375660082009
Tester Steve is doing task with id 0and testcase when description 0 get result 0.8571375660082009
Designer Alycia is doing task  with taskId1 and description description 1
Programmer John is doing task  with task id1 and designLinkhttps://project/design_link_for_task_with_id 1
and testcase when description 1 get result 0.5685295116362588
Tester Steve is doing task with id 1and testcase when description 1 get result 0.5685295116362588
Designer Alycia is doing task  with taskId2 and description description 2
Programmer John is doing task  with task id2 and designLinkhttps://project/design_link_for_task_with_id 2
and testcase when description 2 get result 0.10862379130656497
Tester Steve is doing task with id 2and testcase when description 2 get result 0.10862379130656497
Designer Alycia is doing task  with taskId3 and description description 3
Programmer John is doing task  with task id3 and designLinkhttps://project/design_link_for_task_with_id 3
and testcase when description 3 get result 0.20842921627687072
Tester Steve is doing task with id 3and testcase when description 3 get result 0.20842921627687072
Designer Alycia is doing task  with taskId4 and description description 4
Programmer John is doing task  with task id4 and designLinkhttps://project/design_link_for_task_with_id 4
and testcase when description 4 get result 0.3034475439899075
Tester Steve is doing task with id 4and testcase when description 4 get result 0.3034475439899075
Designer Alycia is doing task  with taskId5 and description description 5
Programmer John is doing task  with task id5 and designLinkhttps://project/design_link_for_task_with_id 5
and testcase when description 5 get result 0.7547273007856262
Tester Steve is doing task with id 5and testcase when description 5 get result 0.7547273007856262
Designer Alycia is doing task  with taskId6 and description description 6
Programmer John is doing task  with task id6 and designLinkhttps://project/design_link_for_task_with_id 6
and testcase when description 6 get result 0.3220132325538253
Tester Steve is doing task with id 6and testcase when description 6 get result 0.3220132325538253
Designer Alycia is doing task  with taskId7 and description description 7
Programmer John is doing task  with task id7 and designLinkhttps://project/design_link_for_task_with_id 7
and testcase when description 7 get result 0.6434931218695579
Tester Steve is doing task with id 7and testcase when description 7 get result 0.6434931218695579
Designer Alycia is doing task  with taskId8 and description description 8
Programmer John is doing task  with task id8 and designLinkhttps://project/design_link_for_task_with_id 8
and testcase when description 8 get result 0.042841051495050464
Tester Steve is doing task with id 8and testcase when description 8 get result 0.042841051495050464
Designer Alycia is doing task  with taskId9 and description description 9
Programmer John is doing task  with task id9 and designLinkhttps://project/design_link_for_task_with_id 9
and testcase when description 9 get result 0.6202169322275158
Tester Steve is doing task with id 9and testcase when description 9 get result 0.6202169322275158
Exception in thread "main" java.lang.IllegalArgumentException: task can't be handled
    at EmployeeChain.doTask(EmployeeChain.java:19)
    at EmployeeChain.doTask(EmployeeChain.java:17)
    at EmployeeChain.doTask(EmployeeChain.java:17)
    at Main.main(Main.java:14)
```

Как видите, Дизайнер берет задачу с айди 0, добавляет к ней дизайн и тесткейсы, обновляет ее в фабрике и потом ее берет уже программист и пишет код и добавляет ссылку на сборку. После обновления в фабрике мы в цикле получаем эту же задачу, дизайнер смотрит на статус, а там готово к тестированию и отдает второму в цепи: программисту, тот отдает тестировщику. Тестировщик обработал и вернул в фабрику. Все. Фабрика в методе получения задачи выдает вторую задачу. (стоит сказать что дизайнер отдает задачу программисту лишь в коде, в реальной жизни все само работает через канбан если вы все настроили правильно).

Все вроде бы хорошо, но вам не кажется что что-то не так? Дизайнер мог бы брать все задачи последовательно и не ждать пока тестировщик закончит работу. Как это сделать?

И это очень хороший вопрос. Давайте для начала уберем выброс исключения

Мы можем это сделать поменяв метод doTask чтобы он вернул boolean.

```
*/
public class EmployeeChain {

    private final Employee employee;
    private EmployeeChain nextEmployeeChain;

    public EmployeeChain(Employee employee) {
        this.employee = employee;
    }

    public boolean doTask(Task task) {
        if (task.getStatus() == employee.getTaskStatus()) {
            employee.doTask(task);
            return true;
        } else if (nextEmployeeChain != null) {
            return nextEmployeeChain.doTask(task);
        } else {
            return false;
        }
    }
```

Если мы можем обработать задачу то вернем true, если нет то вернем результат следующего звена. Если же нет следующего звена, то вернем false. Применение еще проще в мейн методе

```
while (true) {
    if (!chain.doTask(factory.getTask()))
        break;
}
```

Как только мы не можем выполнить задачу завершаем цикл и все.

```
Designer Alycia is doing task  with taskId8 and description descri
Programmer John is doing task  with task id8 and designLinkhttps:/
and testcase when description 8 get result 0.37480212461387596
Tester Steve is doing task with id 8and testcase when description
Designer Alycia is doing task  with taskId9 and description descri
Programmer John is doing task  with task id9 and designLinkhttps:/
and testcase when description 9 get result 0.8662117456639046
Tester Steve is doing task with id 9and testcase when description

Process finished with exit code 0
```

Как видите успешное завершение программы намного лучше чем выход из программы с исключением.

Я бы хотел немного улучшить наш код в том месте, где мы проверяем что статус задачи равен статусу задачи работника который он может взять в работу. Вам не кажется что это слишком низкоуровневый код? Так не должно быть.

```
protected Employee(TaskProgressCallback callback,
                   String name,
                   Task.Status taskStatus) {
    this.callback = callback;
    this.name = name;
    this.taskStatus = taskStatus;
}

public void doTask(Task task) {
    System.out.println(getClass().getSimpleName() + " " + name
            + " is doing task " + getDetails(task));
    callback.updateTask(getTaskWhenDone(task));
}

public boolean canHandleTask(Task task) {
    return taskStatus == task.getStatus();
}
```

Вместо того чтобы отдавать наружу поле класса работника мы перепишем метод таким образом, чтобы метод принимал задачу и отдавал логический результат - равен ли статус задачи статусу задачи работника который он может сделать. И посмотрите как сразу вызов метода улучшил код в классе цепи работника

```
public boolean doTask(Task task) {
    if (employee.canHandleTask(task)) {
        employee.doTask(task);
        return true;
    } else if (nextEmployeeChain != null) {
        return nextEmployeeChain.doTask(task);
    } else {
        return false;
    }
}
```

Вот это уже ООП подход. Если работник может сделать задачу, то пусть делает, если же нет, то пусть отдаст ее другому. Низкоуровневый код где сравнивают 2 обьекта должен быть внутри класса, а не отдавать наружу поле и делайте с ним что хотите.

И давайте вернемся к нашей глобальной задаче. У нас дизайнер берет задачу, после нее ее берет программист и потом тестировщик. Получается что пока программист и тестировщик не закончат свои задачи наш дизайнер не сможет приступить к следующей задаче. Это не совсем хорошо. Дизайнер должен делать задачи одна за другой и не ждать никого. Появилась задача – берем в работу. Вся проблема в том, что у нас один массив для задач. Его нельзя

изменить. В нем хранятся все задачи всех уровней. Вот бы было классно иметь 3 массива с задачами. Но опять же, массивы слишком неудобны. У них конкретный размер и он не может меняться. А удалить задачу из массива нельзя, там будет null. Нам нужно нечто получше чем простой массив. И ответ на этот вопрос будет в следующей лекции. Мы разделим наш массив задач на 3 коллекции под каждый статус задачи и полностью сымитируем реальную жизнь.

В реальности дизайнер получает уведомление когда в столбец с его задачами добавляется новая. Это делается автоматически например в таких штуках как Jira. т.е. следующий работник не назначается руками, а просто задача меняет свой статус и система назначает исполнителя сама. Когда дизайнер закончит работу, то он просто поменяет статус и задача из первой колонки/столбца уйдет в столбец направо. Там система назначит исполнителя программиста.

В следующей лекции мы напишем такой код, чтобы все это работало по воздуху.

А пока перечитайте еще раз лекцию и закрепите ее. Все вопросы можно задать в чате канала.

И на последок давайте уберем множественный return

```
public boolean doTask(Task task) {
    boolean result = false;
    if (employee.canHandleTask(task)) {
        employee.doTask(task);
        result = true;
    } else if (nextEmployeeChain != null) {
        result = nextEmployeeChain.doTask(task);
    }
    return result;
}
```

По возможности пишите 1 return в методе

Но все же мне не нравится классический механизм цепочки обязанностей. Суть в том, что для последней цепи не остается следующего звена и приходится проверять на null. Но как этого избежать? Я придумал как. Давайте посмотрим.

Для начала напишем интерфейс для того чтобы делать задачу

```
public interface TaskHandler {

    boolean doTask(Task task);
}
```

Флаг вернет успех если задачу взяли в работу.

Теперь давайте имплементировать этот интерфейс нашему работнику.

Что поменялось? Метод просто делал задачу, а теперь он проверит сначала сможет ли сделать задачу и если да, то делает ее и вернет флаг. Заметьте мы убрали паблик метод проверки внутрь метода делать задачу и убрали его. Пусть вас не смущают знаки равно.

```java
public abstract class Employee implements TaskHandler {

    private final TaskProgressCallback callback;
    private final String name;
    private final Task.Status taskStatus;

    protected Employee(TaskProgressCallback callback,
                       String name,
                       Task.Status taskStatus) {
        this.callback = callback;
        this.name = name;
        this.taskStatus = taskStatus;
    }

    public boolean doTask(Task task) {
        boolean canHandle = taskStatus == task.getStatus();
        if (canHandle) {
            System.out.println(getClass().getSimpleName() + " " + name
                    + " is doing task " + getDetails(task));
            callback.updateTask(getTaskWhenDone(task));
        }
        return canHandle;
    }

    protected abstract Task getTaskWhenDone(Task task);

    protected abstract String getDetails(Task task);
}
```

Теперь, у нас работник имплементирует интерфейс в котором можно делать задачу, а еще наша новая цепочка работников будет имплементировать этот интерфейс. Посмотрите

```java
public class EmployeeChain implements TaskHandler {

    private final TaskHandler first;
    private final TaskHandler second;

    public EmployeeChain(TaskHandler first,
                         TaskHandler second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public boolean doTask(Task task) {
        boolean result;
        result = first.doTask(task);
        if (!result) {
            result = second.doTask(task);
        }
        return result;
    }
}
```

Ранее мы требовали в конструктор 1 работника, а теперь мы требуем 2 интерфейса. Это может быть как работник, так и цепь работников. В мейне увидете как это работает. Теперь, в методе смотрим так же как и ранее – сможет ли первый участник сделать задачу, если нет, то пусть ее делает второй участник. Самое интересное, что здесь мы можем комбинировать и работников и цепь из работников. Давайте посмотрим на мейн метод

```
public static void main(String[] args) {
    TaskFactory factory = new TaskFactory();
    TaskProgressCallback callback = new CallbackImpl(factory);

    EmployeeChain chain = new EmployeeChain(
            new EmployeeChain(
                    new Designer(callback, name: "Alycia"),
                    new Programmer(callback, name: "John")
            ),
            new Tester(callback, name: "Steve")
    );

    while (true) {
        if (!chain.doTask(factory.getTask()))
            break;
    }
}
```

Мы создаем цепочку обязанностей. Первой цепочкой будет связка из дизайнера и программиста, а вторым звеньем будет лишь тестировщик. Самое интересное в том, что вы можете написать иначе

```
EmployeeChain chain = new EmployeeChain(
        new Designer(callback, name: "Alycia"),
        new EmployeeChain(
                new Programmer(callback, name: "John"),
                new Tester(callback, name: "Steve")
        )
);
```

Результат будет тем же. Самое классное еще в том, что нам не нужно вызывать сеттеры и проверять на null. Наши конструкторы получают сразу 2 интерфейса. Так что для ситуации с несколькими участниками это более правильное на мой взгляд решение.

Вообще можно было бы написать все это и без классов цепочек и интерфейсов, но это был бы низкоуровневый код. С циклом где ищем подходящего исполнителя задачи.

Поэтому мы будем учиться использовать ООП на полную и писать как можно меньше низкоуровнего кода. Да, если ваш код состоит из циклов и условий, то он низкоуровневый. А если он состоит из вызовов методов классов и интерфейсов, то это высокоуровневый код. В нашем мейн методе сначала идет высокоуровневый код, и лишь в конце цикл. Но можно создать класс работа и вызвать метод у него. Но для этого нужно придумать хорошее имя классу и методу.