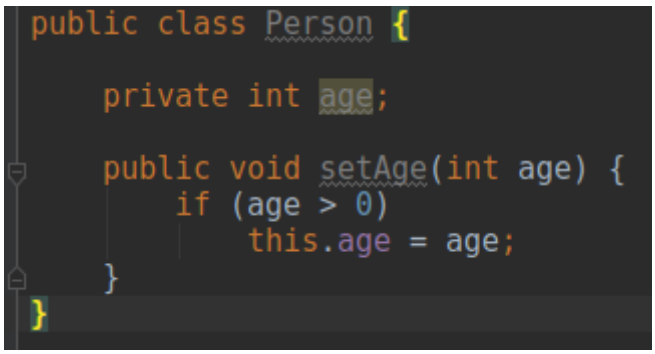
Defensive copy, lists

Передозировка сахаром

Содержание

1. Геттеры и сеттеры, Defensive copy Java

Для начала давайте рассмотрим такой код на джава



Предположим мы имеем дело с таким случаем, когда возраст человека не самое важное в классе и мы не получаем его в конструкторе и поле не final. Для этого мы пишем сеттер поля. Но чтобы это не было бессмысленно пишем хоть какую-нибудь проверку (на деле можно усложнять до проверки верхней грани, но не суть сейчас). Можете добавить else и бросить исключение, неважно. Запомните этот код. Мы вернемся к этому во второй части.

Теперь давайте добавим список чего-то человеку и дадим доступ к списку извне. Пусть у нас будет метод добавления элемента в список и метод геттер для получения списка. Давайте я вам покажу чем это плохо. Переопределим метод toString чтобы понимать содержание обьекта.

Итак, что же не так с нашим методом геттера списка? Во-первых мы отдаем наружу список обьекта и это уже плохо. Но предположим что такое уже есть или нужно сделать и этого не избежать. А теперь посмотрите как легко и просто мы меняем содержимое списка. Да, поле типа список и оно неизменяемое. Т.е. сама ссылка на список не может быть переопределена, но само содержимое может быть изменено легко и просто.

Сначала кладем одни данные в список, после меняем на другие и как мы поменяли содержимое обьекта.

```java
public class Person {

    private int age;
    private final List<String> items = new ArrayList<>();

    public void setAge(int age) {
        if (age > 0)
            this.age = age;
    }

    public void addItem(String item) {
        items.add(item);
    }

    public List<String> getItems() {
        return items;
    }

    @Override
    public String toString() {
        return "Person{" +
                "items=" + items +
                '}';
    }
}
```

```java
    public static void main(String[] args) {
        Person person = new Person();
        person.addItem("one");
        person.addItem("two");
        person.addItem("three");
        System.out.println(person.toString());

        person.getItems().set(0, "new");
        person.getItems().set(1, "new");
        person.getItems().set(2, "new");
        System.out.println(person.toString());

    }
}
```

MainJava › main()

n: MainJava (2) ×

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java
Person{items=[one, two, three]}
Person{items=[new, new, new]}

Process finished with exit code 0
```

Как же это исправить? Самое простое – не давать наружу список. Но есть и другое решение. Оно называется Defensive copy. Когда наружу отдаем не само поле, а его копию. Давайте поменяем геттер и проверим мейн еще раз.

```
public List<String> getItems() {
    return new ArrayList<>(items);
}
```

Просто отдаем наружу новый обьект который имеет в себе список класса.

```
/usr/lib/jvm/java-1.8.0-openjdk-amd
Person{items=[one, two, three]}
Person{items=[one, two, three]}

Process finished with exit code 0
```

В мейне не меняли ничего, но как видите поля обьекта надежно сохранены. Потому что там где мы получали геттером список мы брали не сам список, а копию и там меняли данные, но на сам класс это никак не влияло.

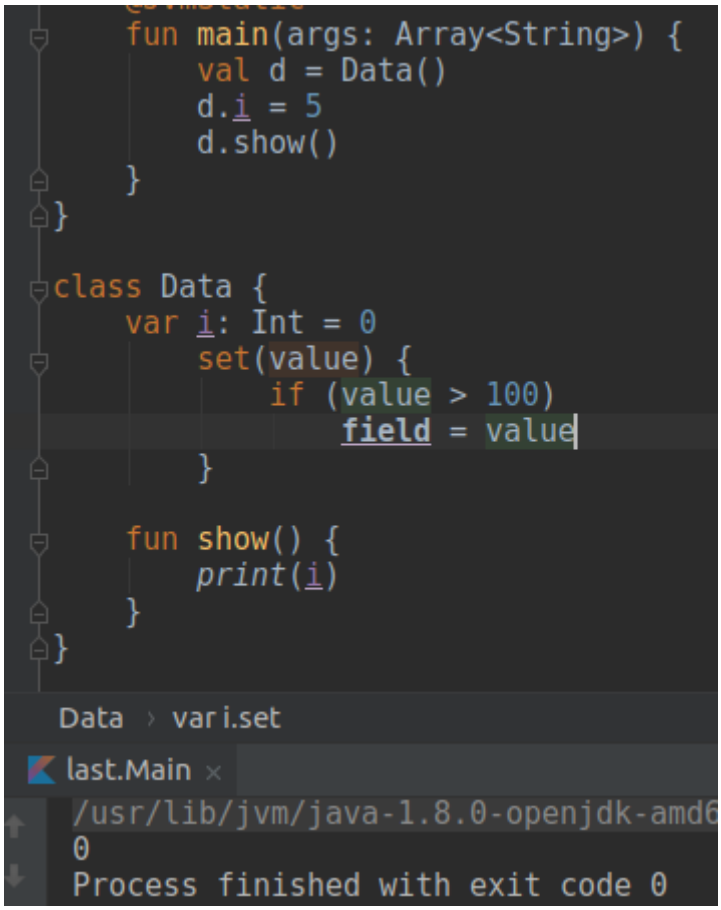
2. Property get set Kotlin

Теперь посмотрим как это все делается в котлин. Если помните, поля класса котлин автоматически становятся геттером и сеттером.

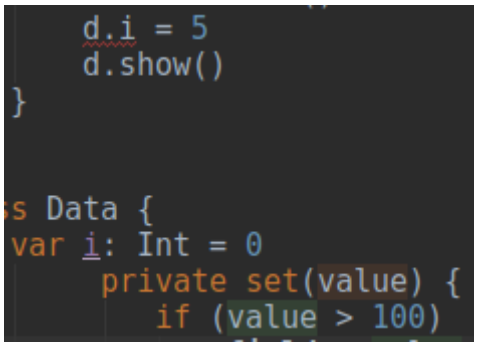
```
object Main {

    @JvmStatic
    fun main(args: Array<String>) {
        val d = Data()
        d.i = 5
        d.show()
    }
}

class Data {
    var i: Int = 0

    fun show() {
        print(i)
    }
}
```

```
Main
last.Main ×
/usr/lib/jvm/java-1.8.0-openjdk-amd
5
Process finished with exit code 0
```

Теперь, мы хотим написать сеттер для поля, но это невозможно? Возможно, но синстаксис иной



Под полем пишем set(value) и то, что будет новым значением придет в value, а чтобы не писать i=value и вызвать рекурсию надо писать field = value вот и все. Как видите при попытке положить в поле число меньше 100 поле не изменилось. Кстати, вы можете в принципе запретить сетить данные



Но самое забавное в том, что можно определить геттер для поля. Посмотрите как забавно. Вы можете сетить в поле что угодно, но при попытке получить значение будете получать всегда одно и то же число Int.MAX_VALUE. Так же можно сделать геттер приватным и наше поле будет доступно лишь внутри. Я конечно же советую всегда писать private val в конструкторе класса, но если вы пишете на котлин, то возможно встретите подобное.

А что насчет списков? В котлин тоже можно сделать Defensive Copy, но намного проще.

```
        val d = Data()
        d.i = 500
        d.show()
    }
}

class Data {
    var i: Int = 0
        set(value) {
            if (value > 100)
                field = value
        }
        get() = Int.MAX_VALUE

    fun show() {
        print(i)
    }
}
```

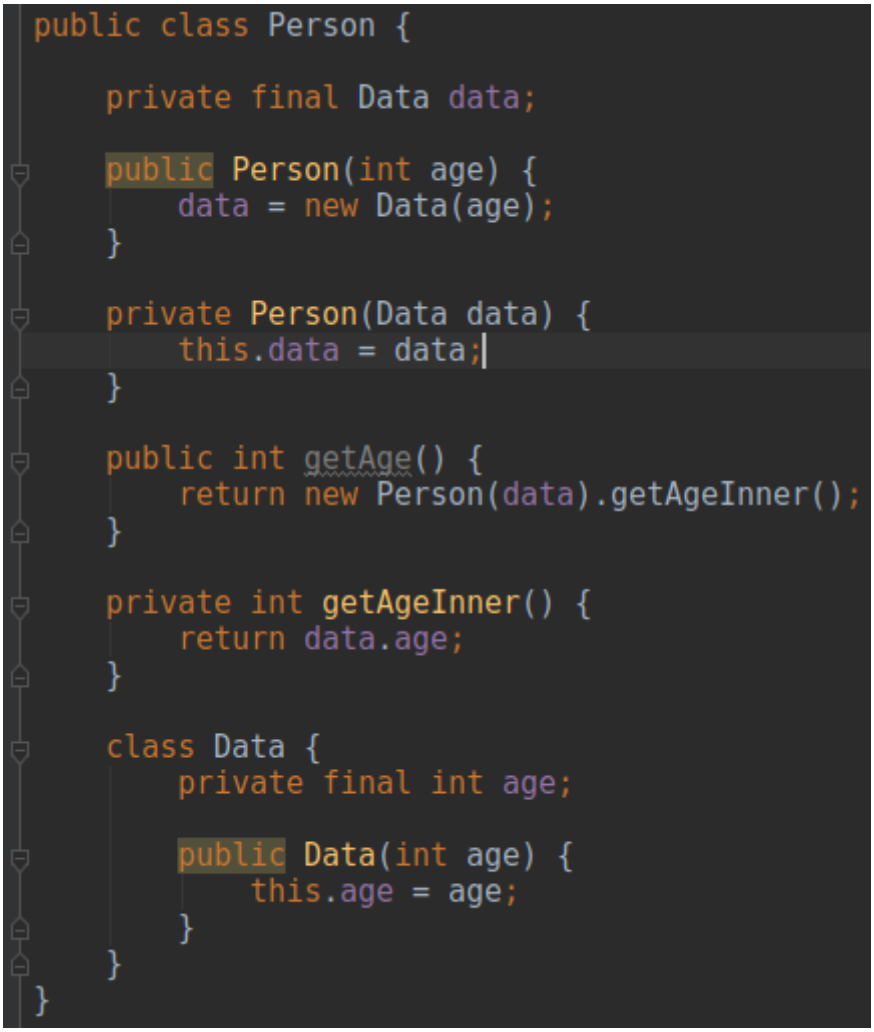
```
Main › main()
last.Main ×
/usr/lib/jvm/java-1.8.0-openjdk-am
2147483647
Process finished with exit code 0
```

В котлин мы делаем так – как видите есть mutableListOf<T> и просто listOf<T>

```
    @JvmStatic
    fun main(args: Array<String>) {
        val d = Data()
        d.add("1")
        d.add("2")
        d.copy[0] = "new"
        d.show()
    }
}

class Data {
    private val list = mutableListOf<String>()

    val copy
        get() = listOf(list)

    fun add(x: String) = list.add(x)


    fun show() {
        print(list)
    }
}
```

Различие в том, что в мутабельном списке можно менять содержимое – добавлять, заменять, удалять элементы. А простой список уже немутабелен – нельзя с ним ничего делать кроме как читать данные. Такой вот простой Defensive copy kotlin style.

Кстати, кто-то может спросить, а как же поступить с другими типами? Со списком понятно. А что делать чтобы другие поля не менялись? Можно вот так



```
public class Person {

    private final Data data;

    public Person(int age) {
        data = new Data(age);
    }

    private Person(Data data) {
        this.data = data;
    }

    public int getAge() {
        return new Person(data).getAgeInner();
    }

    private int getAgeInner() {
        return data.age;
    }

    class Data {
        private final int age;

        public Data(int age) {
            this.age = age;
        }
    }
}
```

Если нужно поле класса, то мы создаем новый обьект и копируем в него данные и отдаем нужное поле.

3. Lists in Kotlin

У списков в котлин много сахара – удобные простые функции для сокращения кода.

Давайте посмотрим на такое – нужно из списка чисел найти все которые больше нуля.

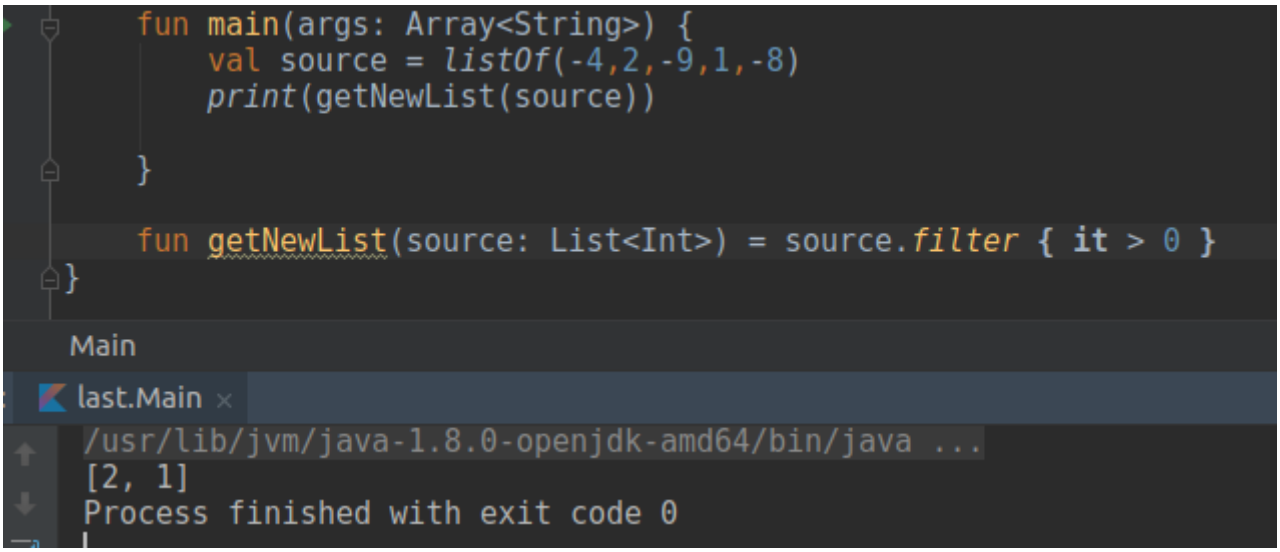
Запомните, если вам нужен список отсортированный по какому-то условию,то не нужно менять исходный список – правильней создать новый список и заполнить его данными.

Простое правило – добавлять лучше чем удалять. Старайтесь по возможности избегать удаления элементов из списка.

А теперь посмотрим на то, как это выглядит в котлин.

```java
private static List<Integer> getNewList(List<Integer> source) {
    List<Integer> positive = new ArrayList<>();
    for (int i : source) {
        if (i > 0) {
            positive.add(i);
        }
    }
    return positive;
}
```

Как видите в котлин намного проще и короче.



Да, ровно 1 линия. А теперь посмотрим на такую задачу – преобразовать список одного вида к другому (да, в андроид мы часто это делаем). В джава будет вот так

```java
private static List<String> getMappedList(List<Integer> source, String prefix) {
    List<String> result = new ArrayList<>();
    for (int i : source) {
        result.add(prefix + i);
    }
    return result;
}
```

То же самое в котлин будет намного проще (в реальности у вас не 1 число и 1 строка, а класс данных в сыром виде и надо привести к финальному виду для юзера).

Так же можно компоновать. Сначала отфильтровать числа по знаку и потом уже преобразовывать.

```
fun main(args: Array<String>) {
    val source = listOf(-4, 2, -9, 1, -8)
    print(getNewList(source, prefix: "num"))
}

fun getNewList(source: List<Int>, prefix: String) = source.map { prefix + it }
}
```

Main › getNewList() › source.map{...}

last.Main

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
[num-4, num2, num-9, num1, num-8]
Process finished with exit code 0
```

Можете посмотреть какие еще есть методы у списков, очень много классных штук.

```
    val source = listOf(-4, 2, -9, 1, -8)
    print(getNewList(source, prefix: "num"))
}

fun getNewList(source: List<Int>, prefix: String) =
    source.filter { it > 0 }.map { prefix + it }
}
```

Main › getNewList()

last.Main

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
[num2, num1]
Process finished with exit code 0
```

Методы first(), last() для получения первого и последнего элементов например.

```
fun main(args: Array<String>) {
    print(listOf(1,2,3,4,5).subList(1,3))
}
```

Main › main()

last.Main

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java
[2, 3]
Process finished with exit code 0
```

Изучите самостоятельно остальные методы. Все их можно найти здесь – kotl.in

4. Введение в coroutines

И на последок давайте скажем пару слов про многопоточность. Для этого рассмотрим старый добрый код на джаве с использованием Thread класса.

Если помните мы скачивали файл с помощью класса наследника Thread и предположим нам надо дать знать чем закончилась операция – успехом или ошибкой.

```java
public class DownloadFile extends Thread {

    private final String url;
    private final String fileName;
    private final ProgressCallback callback;

    public DownloadFile(String url, String fileName, ProgressCallback callback) {
        this.url = url;
        this.fileName = fileName;
        this.callback = callback;
    }

    @Override
    public void run() {
        try (BufferedInputStream in = new BufferedInputStream(new URL(url).openStream());
             FileOutputStream fileOutputStream = new FileOutputStream(fileName)) {
            byte dataBuffer[] = new byte[1024];
            int bytesRead;
            while ((bytesRead = in.read(dataBuffer, i: 0, i1: 1024)) != -1) {
                fileOutputStream.write(dataBuffer, i: 0, bytesRead);
            }
            callback.finishedSuccessfully();
        } catch (IOException e) {
            callback.failed(e.getMessage());
        }
    }

    interface ProgressCallback {

        void failed(String message);

        void finishedSuccessfully();
    }
}
```

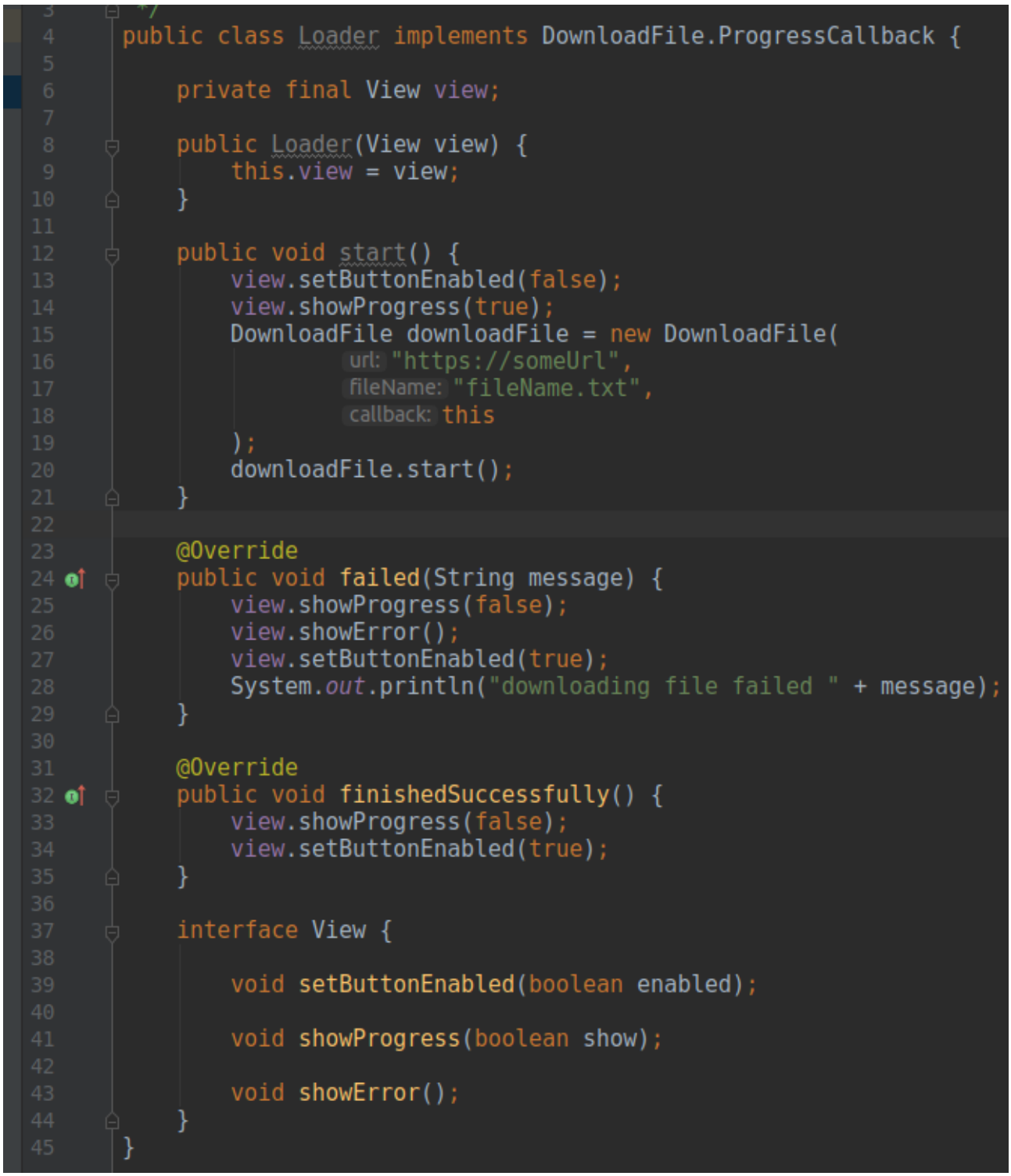
Мы написали простой интерфейс колбека с 2 методами – когда закончили и с каким исходом. Теперь напишем класс, который делает конкретную работу и уже сам меняет на экране телефона что-то.

Пусть наш класс принимает колбек для управления элементами юзер интерфейса и уже внутри делает всю работу. Когда юзер нажмет на кнопку мы захотим не дать ему нажать ее еще раз и покажем загрузчик (прогрессбар). Если загрузка закончится ошибкой, мы ее залогируем, уберем прогресс и покажем стандартную ошибку и вернем возможность юзеру попробовать еще раз. Если же загрузка файла закончится успешно, то мы уберем прогрессбар с экрана и дадим юзеру нажать кнопку еще раз если он захочет. В вашей конкретной программе может быть другая логика. Возможно вы перейдете на другой экран или другие действия будут, тогда вы напишете иной класс Loader и там будет иной интерфейс View, но сам класс DownloadFile остается тем каким он был – загрузка началась и закончилась успехом или ошибкой. Все.

В чем проблема этого подхода? Колбеки.

Код будет выполняться в следующем порядке – 13, 14, 15, 20 и в случае успеха мы перепрыгнем на линию 33, 34 А если будет ошибка, то на 25,26,27,28 И это очень раздражает, потому как ты теряешь нить – где начался код, где он закончился, где продолжение? Почему все так сложно отслеживать? Вам придется поставить 2 брейкпойнта для дебагинга на линиях 25 и 33.

В котлин придумали гениальное решение как обойти эту ситуацию.



```
public class Loader implements DownloadFile.ProgressCallback {

    private final View view;

    public Loader(View view) {
        this.view = view;
    }

    public void start() {
        view.setButtonEnabled(false);
        view.showProgress(true);
        DownloadFile downloadFile = new DownloadFile(
                url: "https://someUrl",
                fileName: "fileName.txt",
                callback: this
        );
        downloadFile.start();
    }

    @Override
    public void failed(String message) {
        view.showProgress(false);
        view.showError();
        view.setButtonEnabled(true);
        System.out.println("downloading file failed " + message);
    }

    @Override
    public void finishedSuccessfully() {
        view.showProgress(false);
        view.setButtonEnabled(true);
    }

    interface View {

        void setButtonEnabled(boolean enabled);

        void showProgress(boolean show);

        void showError();
    }
}
```

Но на данном этапе рассказывать о ней может быть затруднительным для понимания, потому мы вернемся к корутинам в разделе андроид.

Дальнейшее изучение котлина оставляю вам. Можете изучить на офф.сайте котлина kotl.in или же аналогичных ресурсах. Но все базовые особенности котлина мы рассмотрели. Про корутины расскажу ближе к вопросам многопоточности в Андроид.

Но суть вкратце – можно писать асинхронный код таким образом будто он синхронный, т.е. вызывать асихнронный код на линии 1 и на лини 2 сразу же получить результат.