

Множественное наследование

Обходим в Java и обходим в Kotlin

Содержание

1. Классический обход множественного наследования в Java
2. Наследование в Kotlin

1. Классический обход множественного наследования в Java

Для начала давайте еще раз обсудим проблему множественного наследования. Вот у вас есть 2 класса и у каждого свои методы. Вам нужен третий класс, который бы мог делать все то же самое что и оба эти класса. Для этого нужно просто дать в конструктор третьему оба эти класса.

```
interface A {  
    fun doOne()  
    fun doTwo()  
}  
  
interface B {  
    fun doThree()  
}  
  
class AImpl : A {  
    override fun doOne() {  
        print("one")  
    }  
    override fun doTwo() {  
        print("two")  
    }  
}  
  
class BImpl : B {  
    override fun doThree() {  
        print("three")  
    }  
}
```

Давайте посмотрим на обход множественного наследования в Джава для начала.

Напишем класс C на языке джава и посмотрим как это делается

```

public class C implements A, B {

    private final A a;
    private final B b;

    public C(A a, B b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void doOne() {
        a.doOne();
    }

    @Override
    public void doTwo() {
        a.doTwo();
    }

    @Override
    public void doThree() {
        b.doThree();
    }
}

```

Так как мы написали через интерфейсы то можем имплементировать классу C и A и B и просто вызывать нужные методы у нужных классов. Все просто. Посмотрите на применение.

```

public static void main(String[] args) {
    C c = new C(new AImpl(), new BImpl());
    c.doOne();
    c.doTwo();
    c.doThree();
}

```

C

C x

/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java
onetwothree
Process finished with exit code 0

Вуаля, у нашего класса методы разных классов как будто мы наследовались и от A и от B.

В любом случае используйте интерфейсы и все будет проще. Нежели если у вас классы абстрактные без интерфейсов. Тогда малейшее изменение в одном месте порушит все в других местах.

Так же у вас не будет конфликта если в классах A и B будут одинаковые поля. Об этом ниже.

2. Наследование в Котлин

Если вы ожидали чуда от котлина, то зря. Множественное наследование в том виде в котором мы рассмотрели выше недоступно как в джава так и в котлин. Потому что в нашем примере во-первых нет методов с одинаковыми сигнатурами, так же нет полей с одинаковыми типами и именами. Представьте такую ситуацию.

```
interface A {  
    fun doOne()  
}  
  
interface B {  
    fun doOne()  
}  
  
class AImpl(private val i :Int) : A {  
    override fun doOne() {  
        print("oneA $i")  
    }  
}  
  
class BImpl(private val i:Int) : B {  
    override fun doOne() {  
        print("oneB $i")  
    }  
}
```

Итак у вас совпадают как поля классов так и имена методов. Если бы мы могли наследовать оба этих класса, то как бы мы решали конфликты? Вот вы вызываете метод doOne, как он должен отработать? Взять первую реализацию из A или вторую из B? Какое поле взять – из первого класса или второго? Или может оба подряд? Слишком сложно.

Давайте все же попробуем написать класс который наследуется от обоих, посмотрим что будет говорить нам идея

```
class C : AImpl()  
This type is final, so it cannot be inherited from
```

Во-первых еще при наследовании от A мы уже имеем проблему. Класс AImpl финальный. Да, я забыл сказать – в Java если ваш класс помечен final class A то от него нельзя наследоваться. В котлин все классы по умолчанию final если вы руками не дали доступ к наследованию. Вы знаете что делать – Alt+Enter. Теперь класс open т.е. открыт к расширению. Значит его можно наследовать. Теперь обозначим второй класс открытым и посмотрим что будет.

```

open class AImpl(private val i :Int) : A {
    override fun doOne() {
        print("oneA $i")
    }
}

class BImpl(private val i:Int) : B {
    override fun doOne() {
        print("oneB $i")
    }
}

class C : AImpl()

```

Как видите только от 1 класса можно наследоваться

```

class C(private val i:Int) : AImpl(i), BImpl(i)

```

Only one class may appear in a supertype list

Идея предлагает удалить BImpl. Но это не конец! Мы ведь можем наследоваться от 1 класса и любого количества интерфейсов как и в джава, так? Давайте попробуем!

```

class C(private val i:Int) : AImpl(i), B {}

```

Да, в котлин мы уже не пишем длинные слова extends implements. Вместо этого через запятую после : перечисляем всех от кого наследуемся. Как видите никто не указывает нам о том, что у нас не реализован метод в интерфейсе B, потому что реализация уже есть в классе AImpl. Проверим?

```

public static void main(String[] args) {
    C c = new C(1);
    c.doOne();
}

```

un: MainJava x

/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java

oneA 1

Process finished with exit code 0

Да, все так. Как мы можем переопределить так, чтобы вызывался наш новый код для этого метода? Легко

```
class C(private val i:Int) : AImpl(i), B {  
    override fun doOne() {  
        println("hello from C $i")  
    }  
}  
  
C > C() > i  
MainJava x  
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/jav  
hello from C 1  
  
Process finished with exit code 0
```

Хорошо, а что если я не хочу терять реализацию из класса AImpl? Может тогда вызвать super? Пробуем

```
class C(private val i:Int) : AImpl(i), B {  
    override fun doOne() {  
        super.doOne()  
        println("hello from C $i")  
    }  
}  
  
C > doOne()  
MainJava x  
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/jav  
oneA lhello from C 1  
  
Process finished with exit code 0
```

И это все классно. Но я забыл вам рассказать о том, что в Java 8 есть дефолтные методы у интерфейса. Если помните у метода интерфейса не может быть реализации – ну вот в 8 версии джавы решили добавить дефолтное поведение. Давайте напишем интерфейс B на джава с дефолтным методом.

```
interface B {  
    default void doOne() {  
        System.out.println("default B call");  
    }  
}
```

И теперь посмотрим что произошло с нашим классом C в котлин

Как видите конфликт! Слишком много доступных реализаций метода доступно для класса. И сразу же дает решение проблемы – укажите super<Класс> для которого хотите вызвать родительский метод. Давайте попробуем!

```
22
23 class C(private val i:Int) : AImpl(i), MainJava.B {
24
25     override fun doOne() {
26         super.doOne()
27         println("hello from C $i")
28     }
29 }
```

Many supertypes available, please specify the one you mean in angle brackets, e.g. 'super<Foo>'

Как видите я вызвал все методы – сначала у класса A, потом у интерфейса B (я его положил внутрь джавакласса). И как видите все правильно работает.

```
class C(private val i:Int) : AImpl(i), MainJava.B {
    override fun doOne() {
        super<AImpl>.doOne()
        super<MainJava.B>.doOne()
        println("hello from C $i")
    }
}
```

MainJava x

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
oneA ldefault B call
hello from C 1

Process finished with exit code 0
```

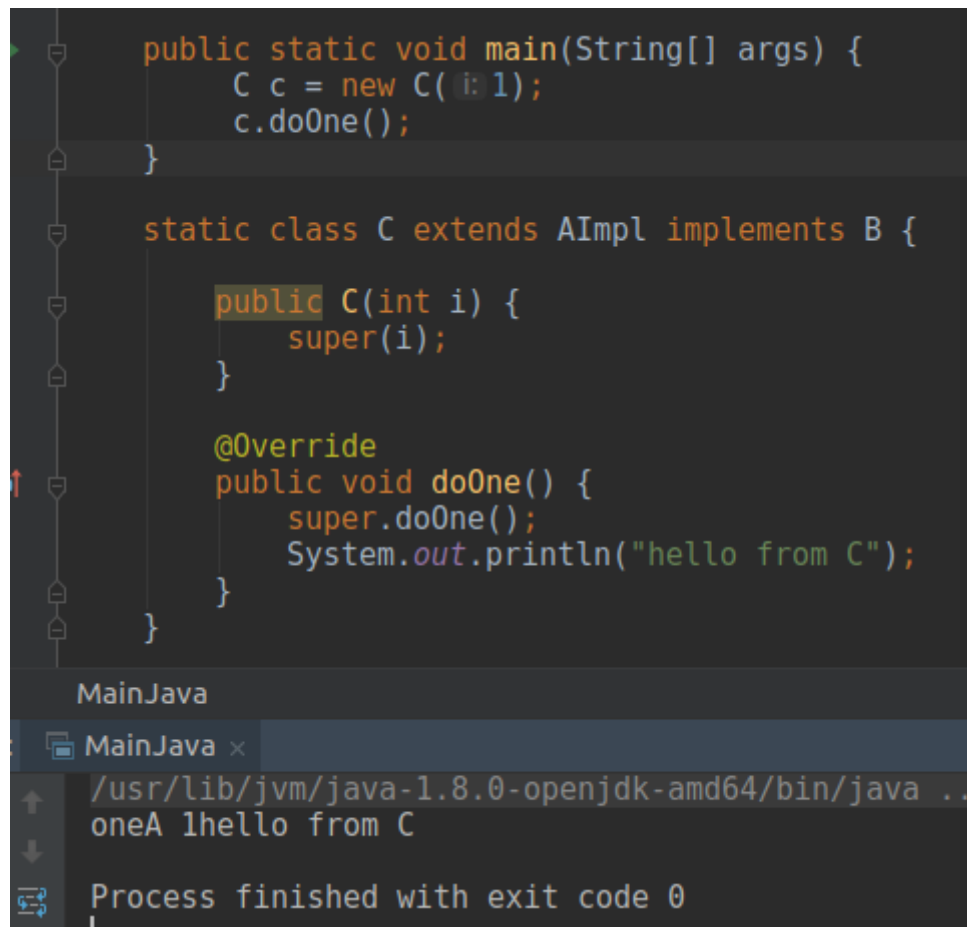
Конечно же я вам не рекомендую в принципе наследоваться от разных классов и интерфейсов, но если у вас все же такая ситуация – просто изучите внимательно все классы и интерфейсы. Кстати, насчет дефолтного метода у интерфейса – в джава 8 версии оно лишь добавилось, а что насчет котлин?

```
internal interface B {
    fun doOne() {
        println("default B call")
    }
}
```

В котлин сразу доступно это. Потому не злоупотребляйте этим. Лучше иметь методы без реализаций чтобы заставить наследников определить их. Или же если вам не обязательно во всех наследниках определять можете написать реализацию в интерфейсе и не писать пустых реализаций в наследниках (помните кейс когда у нашего датасорса было 2 метода, второй использовался лишь в 1 месте из 3?).

Кстати, что же тогда будет в джава в таком случае?

А посмотрите сами – мы не сможем указать от кого наследоваться и потому получим только вызов суперкласса, т.е. интерфейс не является классом и потому super относится лишь к классу.



```
public static void main(String[] args) {
    C c = new C(1);
    c.doOne();
}

static class C extends AImpl implements B {

    public C(int i) {
        super(i);
    }

    @Override
    public void doOne() {
        super.doOne();
        System.out.println("hello from C");
    }
}
```

MainJava

MainJava x

/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ..
oneA lhello from C

Process finished with exit code 0

Так что дефолтная реализация вашего метода в джава мире просто не будет вызвана если сигнатура метода у класса совпадает с сигнатурой метода в интерфейсе от которых наследуется класс.

И это не все плюсы в котлин. Так же есть как и в джава поле класса которое нужно переопределить (напишите сами класс в котором поле val и наследуйтесь). Его можно будет получить как в конструкторе как в джава так и в поле наследника. Но не это самое интересное. А то, что в интерфейсах котлина можно хранить переменные!

Как видите различие между классом и интерфейсом еще меньше в котлин чем в джаве. Теперь вы можете спокойно наследоваться от множества интерфейсов в котлин и тем самым побороть проблему множественного наследования. Ведь в чем было различие абстрактного класса и интерфейса в джава? Что у абстрактного класса могут быть методы с реализацией и поля в классе. А чем же тогда абстрактный класс в джава отличается от интерфейса в котлин, в котором и поля можно иметь и методы с реализацией?

```

interface A {
    val data: String

    fun doOne() {
        print(data)
    }
}

open class AImpl(private val i :Int, override val data: String) : A {
    override fun doOne() {
        super.doOne()
        print("oneA $i")
    }
}

```

Посмотрите на это

```

interface A {
    val data: String

    fun doOne() {
        println(data)
    }
}

interface B {
    val data: String

    fun doOne() {
        println(data + data)
    }
}

class C(override val data: String) : A, B {
    override fun doOne() {
        super<A>.doOne()
        super<B>.doOne()
        println("do one C $data")
    }
}

```

2 идентичных интерфейса A и B хранят поле и метод с одинаковыми сигнатурами. И мы пишем класс C с реализацией их обоих. Что же будет в консоли?


```
public class MainJava {  
    public static void main(String[] args) {  
        C c = new C( data: "text");  
        c.doOne();  
    }  
}  
  
MainJava x  
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/jav  
text  
texttext  
do one C text  
  
Process finished with exit code 0
```

И опять же – мы обошли множественное наследование в Kotlin, но здесь одно но. Наши интерфейсы могут хранить как поля так и методы с реализацией, но они сами не могут наследоваться от классов (от других интерфейсов да) и потому не совсем полноценное множественное наследование.

```
interface A {  
    val data: String  
    fun doOne() {  
        println(data)  
    }  
}  
  
interface B : A {  
    override val data: String  
    override fun doOne() {  
        super.doOne()  
        println(data + data)  
    }  
}  
  
class C(override val data: String) : B {  
    override fun doOne() {  
        super.doOne()  
        println("do one C $data")  
    }  
}
```

И напоследок – если хотите чтобы ваш класс можно было наследовать то пишете open так же работает и с методами – все по умолчанию закрытые – т.е. final, если хотите наследование включить для них то пишете open fun. Вот и все.