

Множественное наследование

Интерфейсы и дженерики

Содержание

1. Множественное наследование в Java
2. Интерфейсы
3. Дженерики

1. Множественное наследование в Java

До сих пор мы рассматривали примеры немного отдаленные от реальной разработки. Возможно кому-то в реальной работе понадобятся треугольники и прямоугольники, но вообще говоря если мы смотрим в мобильную разработку, то там немного другие задачи.

Давайте рассмотрим самую типичную задачу, которую решают мобильные разработчики наверно каждый день. Задача такая – получить из сети данные, закешировать их и исходя из этого получать новые данные из сети или же при неудаче брать из кеша (сохранено во внутреннюю память устройства).

Теперь. У нас будет 2 класса – один для получения данных из сети и второй для получения данных (тех же данных) из кеша. Кто-то может сразу понять – ага, если у обоих классов будет одинаковый метод получения данных, то нужно вынести в абстрактный класс. Так и есть.

```
public class MyData {  
    private final int id;  
    private final String description;  
  
    public MyData(int id, String description) {  
        this.id = id;  
        this.description = description;  
    }  
  
    @Override  
    public String toString() {  
        return "MyData{" +  
            "id=" + id +  
            ", description='" + description + '\'' +  
            '}';  
    }  
}
```

```
public abstract class DataSource {
    public abstract MyData getData();
}
```

Предположим мы получаем некий класс MyData в абстрактном методе. Для простоты пусть у него будет 2 поля. Теперь давайте напишем 2 реализации этого класса – один будет как бы получать эти данные из сети, а другой получать из кеша. Да, класс мы назвали источник данных.

```
public class CachedDataSource extends DataSource {
    @Override
    public MyData getData() {
        return null;
    }
}
```

Итак, если вы написали новый класс и унаследовались от абстрактного класса у которого 1 абстрактный метод который должен вернуть некий объект, то идея при имплементации пишет дефолтный код где return null; возвращает нул.

Так, стоп. Что это такое? Да, друзья мои, пора нам всем немного углубиться в мир языка Джава.

Помните, когда мы говорили о переменных то сказали что у нас есть некий объект, например книга и у нас есть стол с ящиками. И мы создаем книгу и кладем ее в ящик номер 1 к примеру. Если мы говорим, что у нас должна лежать книга в ящике номер 1, но книги пока что нет (ее просто не принесли, не создали), то у нас в ящике будет пустота, т.е. нул.

Более технически это значит, что мы выделили место в памяти, но пока туда ничего не записали и если мы попытаемся получить объект по этой ссылке до того как его туда положили, то нам компилятор вернет null. Это уникальное ключевое слово в Джава которое означает что по этой ссылке (имя переменной) ничего нет (уже или пока что). Нужно сказать еще что null может быть лишь для классов, примитивы не могут хранить null. Посмотрим на пример!

Вот мы получаем объект класса и выводим его в консоль. Но метод который должен вернуть объект вместо него отдает ничего. И когда мы пытаемся получить его как строку то получаем ошибку. Ведь объекта нет. Нельзя получить описание объекта, которого нет. С ним вообще ничего нельзя сделать.

Подебажим – видим что в переменной data хранится null. А нул это не объект и у него нельзя вызвать метод toString(). Именно поэтому мы получаем исключение и наша программа заканчивается с кодом 1. Т.е. не успешно.

Как теперь исправить ситуацию и чтобы наша программа не вылетала? Простой проверкой ну конечно же. Посмотрим.

```
public static void main(String[] args) {
    MyData data = getMyData();
    print(data.toString());
}

private static MyData getMyData() {
    return null;
}
```

Main > main()

Run: Main x

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

Exception in thread "main" java.lang.NullPointerException
at Main.main(Main.java:5)

Process finished with exit code 1

```
public static void main(String[] args) {
    MyData data = getMyData();
    print(data.toString());
}
```

Main > main()

Debug: Main x

Debugger Console →

Frames Variables

"main"@1 in group "..."

main:5, Main

args = {String[0]@513}

data = null

```
public static void main(String[] args) {
    MyData data = getMyData();
    if (data == null) {
        print("object was null");
    } else {
        print(data.toString());
    }
}

private static MyData getMyData() {
    return null;
}
```

Main

Run: Main x

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

object was null

Process finished with exit code 0

Ладно, а нет ли другого способа? Не могу ли я точно знать что метод может вернуть нул или что метод точно вернет не нул и не нужно будет проверять иф елсом каждый раз? Конечно же есть такой способ – аннотации!

```
@Nullable
private static MyData getMyData() {
    return null;
}
```

Мы можем пометить наш метод аннотацией @Nullable и там где мы вызываем метод надо будет проверить на нул. Но есть и другой вариант. Можно пометить метод @NotNull (@NotNull) и тогда мы будем знать что метод вернет точно не нул и проверка не нужна будет.

```
public static void main(String[] args) {
    MyData data = getMyData();
    print(data.toString());
}

@NotNull
private static MyData getMyData() {
    return new MyData( id: 1, description: "1");
}
```

Аннотации ничего не делают в данном случае, а лишь помогают программисту знать какой результат вернет метод.

И теперь давайте вернемся к нашему случаю. Метод получения данных и из сети и из кеша может вернуть нул. Потому что в кеше может ничего не быть и из сети мы можем не получить ничего. Значим пометим наш метод аннотацией в самом абстрактном классе.

```
public abstract class DataSource {

    @Nullable
    public abstract MyData getData();
}
```

И давайте например из сети будем возвращать не нул, а из кеша нул.

```
public class CloudDataSource extends DataSource {

    @Nullable
    @Override
    public MyData getData() {
        return new MyData( id: 1, description: "description_1");
    }
}
```

Кстати, если у вас нет аннотации над методом то можете добавить ее сами.

```


    */
    public class CachedDataSource extends DataSource {

        @Nullable
        @Override
        public MyData getData() {
            return null;
        }
    }


```

Ладно, мы написали абстрактный класс для получения данных и 2 реализации. Теперь же нам нужен конечный класс, который будет исходя из логики сначала смотреть на один класс и потом уже на другой. И здесь мы вспомним про то, что наследоваться можно от 1 класса. Т.е. наш третий класс не может наследоваться от CachedDataSource и от CloudDataSource одновременно. Что же делать? Давайте посмотрим на следующий код

```


    public class Repository extends DataSource {

        private final DataSource cloudDataSource;
        private final DataSource cachedDataSource;

        public Repository(DataSource cloudDataSource,
                          DataSource cachedDataSource) {
            this.cloudDataSource = cloudDataSource;
            this.cachedDataSource = cachedDataSource;
        }

        @Override
        public MyData getData() {
            MyData result = cachedDataSource.getData();
            if (result == null) {
                result = cloudDataSource.getData();
            }
            return result;
        }
    }


```

Сначала вам покажется это немного сложным. Но давайте разберемся. Класс repository получает в конструктор 2 разных объекта датасурса, один для работы по сети, другой по кешу. И нам все равно нужен метод получения данных, значит наследуемся от абстрактного класса и получаем метод получения данных getData и в нем уже пишем нашу логику.

Сначала мы попробуем извлечь данные из кеша, если там ничего нет, то пойдем в сеть. Вы можете написать обратную логику, все зависит от конкретного проекта. Здесь лишь вопрос приоритета. Предположим у вас данные одни и те же и зачем лезть в сеть каждый раз если у вас уже в кеше все есть. Логично, неправда ли? Но если у вас в сети всегда могут быть данные посвежее, то тогда дергайте метод получения данных из сети сперва и если нет интернет соединения например или ошибка произошла и ничего не вернулось, то смотрим в

кеш. Ладно, теперь у нас есть класс который смотрит в 2 датасорса. Мы написали что у нас из сети есть данные, а в кеше нет их. Попробуем вызвать метод у репозитория.

```
public static void main(String[] args) {
    DataSource repository = new Repository(
        new CloudDataSource(),
        new CachedDataSource()
    );

    MyData data = repository.getData();
    print(data.toString());
}

private static void print(Object text) { System.out.println(text); }
```

Main

in: Main x

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

MyData{id=1, description='description_1'}

Process finished with exit code 0

Вы можете подебажить и увидите, что сначала мы посмотрели в кеш и там ничего было ноль и потом глянули в сеть и оттуда получили объект с айди 1.

Ладно, вроде разобрались (если нет, подебажьте). И здесь у нас встанет вопрос. А какой толк от такого кода в репозитории? Мы получили данные из сети и их же надо ведь положить в кеш, так? А мы каждый раз смотрим в пустой кеш и дергаем сеть каждый раз. Неправильно как-то.

ОК, какие у нас варианты? Нам определенно нужен еще один метод у абстрактного класса датасорса где мы сохраним данные. Давайте же его просто добавим.

```
public abstract class DataSource {

    @Nullable
    public abstract MyData getData();

    public abstract void saveData(@NotNull MyData data);
}
```

Как видите можно помечать аннотацией не только сами методы, но и аргументы метода. Мы будем знать, что пришел объект точно не ноль и сможем его сохранить. Давайте посмотрим на наши реализации!

Для кеш датасорса мы просто положим объект и вернем его обратно в методе. Для этого добавим простую переменную.

```

public class CachedDataSource extends DataSource {

    private MyData myData;

    @Nullable
    @Override
    public MyData getData() {
        return myData;
    }

    @Override
    public void saveData(@NotNull MyData data) {
        myData = data;
    }
}

```

Да, опять же если мы не успели положить ничего, то и это ничего и вернем. Но мы точно знаем что перезапишем переменную на ненул. Ок. Разобрались. Но подождите, если мы внесли изменения в абстрактный класс датасорса, то и во всех наследниках будут изменения, так? Посмотрим на клаудддатасорс

```

public class CloudDataSource extends DataSource {

    @Nullable
    @Override
    public MyData getData() {
        return new MyData( id: 1, description: "description_1");
    }

    @Override
    public void saveData(MyData data) {

    }
}

```

Эм, ну у нас теперь 2 метода и второй абсолютно не нужен. Мы же получаем данные из сети, а никак не сохраняем их... в сеть? Ладно, пускай будет пустой метод здесь. Но... у нас же есть еще третий наследник, так?

Что же у нас вышло? Мы добавили в абстрактный класс еще один метод, который по сути нужен лишь в 1 классе из 3. В этих 2 классах он просто будет пустой и это как бы не супер, не находите? Конечно же можно сделать так – в абстрактном поменять его на пустой и переопределять лишь в 1. Типа

```

public class Repository extends DataSource {

    private final DataSource cloudDataSource;
    private final DataSource cachedDataSource;

    public Repository(DataSource cloudDataSource,
                     DataSource cachedDataSource) {...}

    @Override
    public MyData getData() {...}

    @Override
    public void saveData(MyData data) {

    }
}

```

```

public abstract class DataSource {

    @Nullable
    public abstract MyData getData();

    public void saveData(@NotNull MyData data) {

    }
}

```

И тогда можно убрать пустые методы из 2 других классов – cloudDataSource, repository.

И все же у нас остается маленькая неприятная штука. В мейне, если например нашим классом пользуется другой программист, он сможет вызвать не только метод получения данных, но еще и метод сохранения. Это немного неприятно, потому что конечный программист не должен знать о внутренних хитросплетениях наших классов, что у нас 3 наследника одного класса и один из них умеет кешировать данные.

Так как же решить эту проблему? Было бы классно если бы был способ скрыть некие методы из видимости. И да, на самом деле есть такое – package-private видимость метода вместо public. Тогда метод будет виден лишь в том пакете в котором он есть. Но это не решает более глобально проблему, поэтому давайте оставим это совсем и подумаем над вот чем

было бы классно если бы конкретные классы имели конкретные методы – клауд и репозиторий могли бы только получать данные, а кешдатасоурс еще и сохранять.


```

public static void main(String[] args) {
    DataSource repository = new Repository(
        new CloudDataSource(),
        new CachedDataSource()
    );

    MyData data = repository.getData();
    repository.
    print(d
}

private sta

```

Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>

2. Интерфейсы

Маленький спойлер – здесь мы сразу рассмотрим один из принципов SOLID, но об этих принципах более подробно будет отдельная лекция.

Итак, если у нас есть классы которые мы можем только наследовать и при наследовании должны переопределять все абстрактные методы и все публичные методы видны, то должен существовать способ разделять методы. Для этого и существуют интерфейсы. Думайте о интерфейсах как о недоклассах. Если мы можем в классе хранить какие-то переменные и писать в них методы, то в интерфейсах у нас могут быть только методы, у которых нет реализации. Т.е. интерфейс это как абстрактный класс, у которого нет полей и нет неабстрактных методов.

Но самое важное отличие от класса в том, что любой класс может имплементировать любое количество интерфейсов. Здесь мы обходим ограничение языка Джава на множественное наследование.

Итак, давайте для начала вынесем метод получения данных в отдельный интерфейс. Точнее давайте перепишем класс DataSource и из него сделаем интерфейс. Это делается очень просто. Смотрите:

```
public interface DataSource {

    @Nullable
    MyData getData();

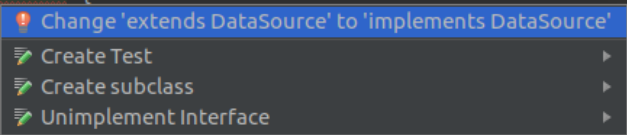
    void saveData(@NotNull MyData data);
}
```

Заметьте, что теперь у интерфейса ключевое слово `interface` вместо `abstract class` и методы без `public` или `protected`. И еще один момент – нет тела у методов, кончается `;`. Хорошо, посмотрим же тогда какие изменения требуются у наследников:

```
public class CachedDataSource extends DataSource {

    private MyData myData;

    @Nullable
    @Override
    public MyData getData() {
```



Нажимаем `Alt+Enter` и получаем:

```
public class CachedDataSource implements DataSource {

    private MyData myData;

    @Nullable
    @Override
    public MyData getData() {
        return myData;
    }

    @Override
    public void saveData(@NotNull MyData data) {
        myData = data;
    }
}
```

Раньше было ключевое слово `extends` после которого должен был следовать класс, а теперь у нас ключевое слово `implements` после которого должен идти интерфейс. В остальном же все как и раньше. Посмотрим на другие реализации.

Так, стоп. Мы опять получили то же что и было раньше – класс у которого пустой метод. Ничего не поменялось, что же мы сделали? Да, мы написали интерфейс вместо абстрактного класса, но мы забыли зачем мы это делали. Для того чтобы разделить методы. Давайте тогда сделаем вот что. Напишем интерфейс в котором будут 2 метода и еще один интерфейс в котором будет 1 метод.

```

public class CloudDataSource implements DataSource {

    @Nullable
    @Override
    public MyData getData() {
        return new MyData( id: 1, description: "description_1");
    }

    @Override
    public void saveData(MyData data) {

    }

}

```

Пусть у нашего датасorsa будет 1 метод для получения данных

```

public interface DataSource {

    @Nullable
    MyData getData();

}

```

А для получения данных и сохранения пусть будет мутабельный (изменяемый) интерфейс с 2 методами

```

public interface MutableDataSource {

    @Nullable
    MyData getData();

    void saveData(@NotNull MyData data);

}

```

И давайте сразу здесь сделаем паузу.

У нас был 1 абстрактный класс в котором 2 метода но не везде они были нужны. Мы сделали 2 интерфейса, в одном 1 метод где будет 2 наследника, и еще один интерфейс где будет 1 конкретный наследник. Хорошо. Если понятно, давайте пойдем дальше.

Нужно теперь имплементировать нужные интерфейсы в наследниках. Посмотрим как теперь будут выглядеть наши классы.

```

public class CloudDataSource implements DataSource {

    @Nullable
    @Override
    public MyData getData() {
        return new MyData( id: 1, description: "description_1");
    }

}

```

Здесь все хорошо – 1 метод у датасорс интерфейса и он имеет реализацию у клауддатасорса.

```
public class CachedDataSource implements MutableDataSource {  
    private MyData myData;  
  
    @Nullable  
    @Override  
    public MyData getData() { return myData; }  
  
    @Override  
    public void saveData(@NotNull MyData data) { myData = data; }  
}
```

Здесь тоже все хорошо. У нас 2 метода в интерфейсе и они оба имеют реализацию в кешдатасорсе. Посмотрим же теперь на репозиторий.

```
public class Repository implements DataSource {  
    private final DataSource cloudDataSource;  
    private final DataSource cachedDataSource;  
  
    public Repository(DataSource cloudDataSource,  
                     DataSource cachedDataSource) {  
        this.cloudDataSource = cloudDataSource;  
        this.cachedDataSource = cachedDataSource;  
    }  
  
    @Override  
    public MyData getData() {  
        MyData result = cachedDataSource.getData();  
        if (result == null) {  
            result = cloudDataSource.getData();  
        }  
        return result;  
    }  
}
```

Так, ну здесь все как и было до этого. Интерфейс датасорса с 1 методом чтобы на выход в мой класс можно было видеть только его. Хорошо. Но зачем мы добавили второй интерфейс со вторым методом? Чтобы его использовать. Для этого надо немного поменять код.

Теперь мы получаем в конструктор один датасорс простой для сети и второй датасорс с возможностью хранить данные, а не только получать. И посмотрите теперь на метод получения данных. Мы пробуем получить данные из кеша, если их нет, то мы берем данные из сети и кладем их в кешдатасорс. В следующий раз когда нам понадобятся данные мы уже получим их из кеша, ведь там будут лежать те данные, которые мы получили первый раз из сети. Чтобы понимать о чем речь, давайте залогируем метод в репозитории.

```

public class Repository implements DataSource {

    private final DataSource cloudDataSource;
    private final MutableDataSource cachedDataSource;

    public Repository(DataSource cloudDataSource,
                     MutableDataSource cachedDataSource) {
        this.cloudDataSource = cloudDataSource;
        this.cachedDataSource = cachedDataSource;
    }

    @Override
    public MyData getData() {
        MyData result = cachedDataSource.getData();
        if (result == null) {
            result = cloudDataSource.getData();
            cachedDataSource.saveData(result);
        }
        return result;
    }
}

```

```

@Override
public MyData getData() {
    MyData result = cachedDataSource.getData();
    if (result == null) {
        print("no data in cache");
        result = cloudDataSource.getData();
        cachedDataSource.saveData(result);
    } else {
        print("getting from cache");
    }
    return result;
}

private void print(String text) {
    System.out.println(text);
}

```

И давайте 2 раза подряд вызовем метод получения данных в мейн классе

Сначала мы пытаемся получить данные из кеша, там ничего нет, мы получили данные из сети и положили в кеш. Во второй раз мы смотрим в кеш и там уже есть данные и мы их получаем. В сеть второй раз не идем. Все просто, неправда ли? Да, данные одни и те же, потому что мы никак не поменяли его в ходе хранения. Мы можем написать как-то поменять их в ходе чтобы было понятно откуда данные, но логирование (когда просто выводишь в консоль что-то) прекрасно с этим справляется.


```
public static void main(String[] args) {
    DataSource repository = new Repository(
        new CloudDataSource(),
        new CachedDataSource()
    );

    MyData data = repository.getData();
    print(data.toString());
    data = repository.getData();
    print(data.toString());
}
```

Main > main()

Output: Main x

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java .
no data in cache
MyData{id=1, description='description_1'}
getting from cache
MyData{id=1, description='description_1'}
Process finished with exit code 0
```

И если вы думаете что на этом все, то нет. Мы нарушили один из принципов программирования, а именно дублирование кода. DRY – don't repeat yourself. Если в вашем проекте есть код, который встречается в точно таком же виде еще один раз, то вы нарушили этот принцип. Но где? Где же нарушение? А я вам скажу где. Посмотрите на наши интерфейсы! В одном интерфейсе 1 метод, а во втором 2 метода, но посмотрите внимательно на один из методов во втором интерфейсе, он же точно такой же как тот метод в первом интерфейсе.

```
public interface DataSource {

    @Nullable
    MyData getData();

}
```

```
public interface MutableDataSource {

    @Nullable
    MyData getData();

    void saveData(@NotNull MyData data);

}
```

Хорошо, метод `MyData getData()` встречается в 2 разных интерфейсах, но как мы можем исправить это? И здесь мы будем использовать интерфейсы на полную катушку. Одна из самых классных фишек языка Джава в том, что можно наследовать интерфейсы. ЧАВО?

```

5
6 public interface MutableDataSource extends DataSource {
7
8     void saveData(@NotNull MyData data);
9
10 }

```

Хоба! Теперь наш интерфейс имеет 1 свой метод — saveData, а тот второй метод, который совпадал с тем же методом из другого интерфейса теперь будет доступен так же как и ранее просто потому что мы наследовали второй интерфейс от первого. Чтобы проверить что ничего не изменилось, просто посмотрите на класс, который имплементировал наш интерфейс. Там все в порядке. Идея ниначто не ругается. Кстати быстро перейти в реализацию интерфейса можно по стрелке вниз слева от объявления интерфейса.

Interface Segregation SOLID

Ладно, мы убрали абстрактный класс в котором было 2 метода, но для 3 наследников второй метод не был нужен и мы разбили 1 класс на 2 интерфейса, в одном один метод, который используется во всех 3 наследниках и второй интерфейс который наследуется от первого интерфейса и имеет свой второй метод который нужен лишь 1 классу.

И все бы прекрасно, все классно, все супер. Но есть одно но. Да что опять? У нас жесткая завязка на наш класс MyData.

3. Дженерики

Давайте предположим что в вашей программе (приложение на андроид например) имеется несколько фичей — несколько экранов, где получаем разные данные. Предположим механизм один и тот же, сначала смотрим в кеш и потом уже идет в сеть, Тогда нам нужно написать 2 интерфейса и 3 класса и лишь поменять наш класс MyData на что-то другое? Согласитесь это слишком. Мы опять нарушим принцип DRY. Было бы классно переиспользовать все классы и интерфейсы но чтобы у нас можно было использовать разные классы данных.

Но ведь наш джава класс компилируется в байт код и из прекрасных слов на английском языке оно все превратится в нули и единицы. Как мы сможем поменять что-то после этого? Никак. Значит нам нужно параметризовать уже написанный код.

Знакомьтесь, дженерики!

Начнем с нашего простого интерфейса.

```

public interface DataSource<T> {

    @Nullable
    T getData();

}

```

Раньше у нас была жесткая завязка на MyData. Теперь же мы убрали этот класс и заменили его дженериком T. Т.е. теперь наш интерфейс может работать с абсолютно любыми данными. Мы просто передадим конкретный класс в аргумент и все будет работать по-прежнему.

Посмотрим же на интерфейс наследник:

```
public interface MutableDataSource<T> extends DataSource<T> {  
    void saveData(@NotNull T data);  
}
```

Мы сохраняем тот же тип класса который у нас был в датасоурсе и потому во время наследования указываем тот же тип T. Ладно, что же будет с наследниками классами?

```
public class CloudDataSource implements DataSource<MyData> {  
    @Override  
    public MyData getData() {  
        return new MyData( id: 1, description: "description_1");  
    }  
}
```

Здесь уже мы передаем конкретный класс в интерфейс и получаем наш метод. Согласен, по-хорошему нужно переименовать класс в MyDataCloudDataSource. Сделайте это сами.

```
public class CachedDataSource<T> implements MutableDataSource<T> {  
    private T myData;  
  
    @Nullable  
    @Override  
    public T getData() {  
        return myData;  
    }  
  
    @Override  
    public void saveData(@NotNull T data) {  
        myData = data;  
    }  
}
```

А здесь уже мы применим дженерик тип T к полю и к методам. Если нам нужен будет кешдатоурс для класса MyData то не нужно его создавать. Просто передадим его в аргумент класса.

И давайте посмотрим как поменялся наш класс репозитория. Здесь мы тоже передаем дженерик тип T во все датасоурсы и оставляем нашу логику получения данных но уже с дженерик типом T. Мы получим какой-то тип данных из кеша, если там нул, то получим из сети и потом уже сохраним в кеш и вернем его.

И чтобы все это проверить нам нужно создать еще один класс и создать 2 репозитория и еще второй датасоурс для сети для второго класса

```

public class Repository<T> implements DataSource<T> {

    private final DataSource<T> cloudDataSource;
    private final MutableDataSource<T> cachedDataSource;

    public Repository(DataSource<T> cloudDataSource,
        MutableDataSource<T> cachedDataSource) {
        this.cloudDataSource = cloudDataSource;
        this.cachedDataSource = cachedDataSource;
    }

    @Override
    public T getData() {
        T result = cachedDataSource.getData();
        if (result == null) {
            print("no data in cache");
            result = cloudDataSource.getData();
            cachedDataSource.saveData(result);
        } else {
            print("getting from cache");
        }
        return result;
    }

    private void print(String text) {
        System.out.println(text);
    }
}

```

```

public class GeoData {

    private final int id;
    private final double longitude;
    private final double latitude;

    public GeoData(int id, double longitude, double latitude) {
        this.id = id;
        this.longitude = longitude;
        this.latitude = latitude;
    }

    @Override
    public String toString() {
        return "GeoData{" +
            "id=" + id +
            ", longitude=" + longitude +
            ", latitude=" + latitude +
            '}';
    }
}

```

Пусть это будет класс для хранения долготы и широты с неким айди

```
public class GeoDataCloudDataSource implements DataSource<GeoData> {  
    @NotNull  
    @Override  
    public GeoData getData() {  
        return new GeoData( id: 12, longitude: 52.12, latitude: 43.98);  
    }  
}
```

Нам нужен конкретный класс для получения геоданных из сети, здесь мы просто вернем объект, но в реальности будет намного больше кода, сами понимаете.

И давайте напишем 1 класс репозитория для геоданных, а для первого типа данных не будем писать, чтобы вы видели разницу.

```
public class GeoRepository extends Repository<GeoData> {  
    public GeoRepository(DataSource<GeoData> cloudDataSource,  
        MutableDataSource<GeoData> cachedDataSource) {  
        super(cloudDataSource, cachedDataSource);  
    }  
}
```

Здесь мы написали класс с конкретным типом и он требует конструктор, где автоматом подставляет нужный тип.

Чтобы понять нужно ли нам писать такие классы или нет, можно посмотреть как это работает в мейн классе.

```
public static void main(String[] args) {  
    DataSource<MyData> myDataDataSource = new Repository<>(  
        new MyDataCloudDataSource(),  
        new CachedDataSource<>()  
    );  
  
    DataSource<GeoData> geoDataDataSource = new GeoRepository(  
        new GeoDataCloudDataSource(),  
        new CachedDataSource<>()  
    );  
    MyData data = myDataDataSource.getData();  
    GeoData geoData = geoDataDataSource.getData();  
    print(data.toString());  
    print(geoData.toString());  
}
```

Сначала мы создаем репозиторий для первого типа данных и просто пишем new Repository как видите передавать второй раз тип MyData не нужно, мы его передали в объявлении

переменной. Внутри уже отдаем нужный класс для работы с сетью и кешдатасорс без явного типа, потому что он берется из объявления переменной.

Но как видите можно сделать то же самое и с георепозиторием, единственное отличие что вместо Repository<> у вас будет конкретный класс GeoRepository, внутри которого все равно передаем геоклауд и кешдатасоурс без типа.

После чего можно уже работать с датасоурсами и получать данные. Сколько угодно раз.

Согласен, немного длинная получилась лекция, но как видите все темы внутри нее связаны очень хорошо и было бы грехом не написать все и сразу.

В следующих лекциях мы подробно остановимся на принципах разработки, рассмотрим конкретные и я объясню почему это так важно.

А пока можете закрепить тему следующей задачей.

Помните работников? Пусть будет абстрактный класс работника который может работать с какими-то задачами и у этого класса наследники Программист, Дизайнер и Тестировщик. Каждый класс сам решает как выполнять задачи. Написать класс задача и наследников 3 для него. Связать класс работника с классом задачи (оба абстрактные, вместо просто Т напишете этот класс) и в методе выполнить задачу заюзаете его. И чтобы усложнить это задание, пусть например дизайнер не только выполняет задачи, но и порождает их.

* супер усложнение для тех кто хочет – пусть будет класс собирающий экстразадания аля фабрика задач и распределяющий задачи по их типу. С этого мы начнем следующую лекцию.

Заметка для меня – тема колбеков и коллекции.