Евгений Гродно

36 уровень

30 апреля 2018 17:45 4351 15

JUnit part I

Пост из группы Random

995223 участников

JUnit :: или как полюбить валидатор JavaRush.

Кратко о том, зачем этот зверь нам нужен? **JUnit** — это фреймворк автоматического тестирования вашего хорошего ~~или не совсем хорошего~~ кода. Можно сказать: — зачем мне эти качели, я и так смогу легко, и просто протестировать свой хороший Java код. Можно много писать вступительной лирики, но поэт из меня никакой, перейдём лучше к делу…

Создаем объект

И так, чтобы что-то тестировать, нам для начала нужен объект тестирования. Перед нами стоит задача.

1. Нам нужен объект, который будет хранить информацию о Пользователе.
    a. Id - нужно считать по порядку добавления нового пользователя.
    b. Имя пользователя.
    c. Его возраст.
    d. Пол (male/female)

2. Нужно предусмотреть хранение списка пользователей.

3. Класс должен уметь.
    a. Формировать список всех пользователей.
    b. Формировать список пользователей по полу (MALE/FEMALE).
    c. Возвращать количество пользователей в общем списке, и посчитать количество по признаку пола пользователя.
    d. Посчитать общую сумму по возрасту пользователей, так же учесть по признаку пола.
    e. Посчитать средний возраст, как общий так и по признаку пола.

И так, приступим к созданию объекта…

Создадим Java класс `User` он будет содержать поля:

```
1  private int id;
2  private String name;
3  private int age;
4  private Sex sex;
```

Для хранения данных о пользователе этого достаточно, посмотрим что там еще нужно по задаче.

Нам нужно как-то хранить всех пользователей, сделаем в нашем классе статическое поле `allUsers`, думаю нормально если это будет `Map<Integer, User>`

```
1  private static Map<Integer, User> allUsers;
```

Еще нам как-то нужно присваивать порядковый номер пользователям, создадим статическое поле счетчик, который при создании нового пользователя, будет присваивать порядковый `Id` пользователю.

```
1  private static int countId = 0;
```

Так, с полями вроде разобрались, напишем конструктор для нашего объекта, и гетеры для полей `id`, `name`, `age`, `sex`. С гетерами там ничего сложного нет, попросим помощи у **IDEA**, она никогда не откажет, а конструктор сделаем немного с хитростью.

Конструктор будет уметь. Инициализировать поля, проверять есть ли такой объект в `allUsers`, если такого объекта нет, то увеличиваем наш счетчик `countId++`, и добавляем его в список всех пользователей. А так же инициализировать поле `allUsers` ели оно еще не было инициализировано.

Для удобства поиска одинаковых объектов, переопределим методы `equals()` и `hashCode()`, опять попросим помощи у любимой **IDEA** и будем сравнивать по полям `name`, `age`, `sex`. Плюс создадим приватный метод `hasUser()`, который будет проверять есть ли такой объект в списке.

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    User user = (User) o;
    return age == user.age &&
            Objects.equals(name, user.name) &&
            sex == user.sex;
}

@Override
public int hashCode() {

    return Objects.hash(name, age, sex);
}
```

Конструктор в итоге у меня получился такой.

```
public User(String name, int age, Sex sex) {
    if (allUsers == null){
        allUsers = new HashMap<>();
    }

    this.name = name;
    this.age = age;
    this.sex = sex;

    if (!hasUser()){
        countId++;
        this.id = countId;
        allUsers.put(id, this);
```

```
    }
}
```

и вспомогательный приватный метод

```
private boolean hasUser(){
    for (User user : allUsers.values()){
        if (user.equals(this) && user.hashCode() == this.hashCode()){
            return true;
        }
    }
    return false;
}
```

а также переопределим `toString()`

```
@Override
public String toString() {
    return "User{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", age=" + age +
            ", sex=" + sex +
            '}';
}
```

Теперь пришло время, реализовать логику требуемых методов. Так как логика в основном будет работать со статическими полями, методы тоже сделаем статическими, для объектов они не нужны.

a. Формировать список всех пользователей.
b. Формировать список пользователей по полу(MALE/FEMALE).

   С пунктами **a** и **b** хорошо справится метод `getAllUsers()` который будет возвращать лист всех `User`, и перегруженный метод `getAllUsers(Sex sex)` он будет возвращать список, в зависимости от переданного параметра `Sex`.

```
public static List<User> getAllUsers(){
    return new ArrayList<>(allUsers.values());
}

public static List<User> getAllUsers(Sex sex){
    List<User> listAllUsers = new ArrayList<>();
    for (User user : allUsers.values()){
        if (user.sex == sex){
```

```
            listAllUsers.add(user);
        }
    }
    return listAllUsers;
}
```

c. Возвращать количество пользователей в общем списке, и посчитать количество по признаку пола пользователя.

```
public static int getHowManyUsers(){
    return allUsers.size();
}

public static int getHowManyUsers(Sex sex){
    return getAllUsers(sex).size();
}
```

d. Посчитать общую сумму по возрасту пользователей, так же учесть по признаку пола. Для этой задачи сделаем методы.

```
public static int getAllAgeUsers(){
    int countAge = 0;
    for (User user : allUsers.values()){
        countAge += user.age;
    }
    return countAge;
}

public static int getAllAgeUsers(Sex sex){
    int countAge = 0;
    for (User user : getAllUsers(sex)){
        countAge += user.age;
    }
    return countAge;
}
```

e. Посчитать средний возраст, как общий так и по признаку пола.

```
public static int getAverageAgeOfAllUsers(){
    return getAllAgeUsers() / getHowManyUsers();
}

public static int getAverageAgeOfAllUsers(Sex sex){

```

```
    return getAllAgeUsers(sex) / getHowManyUsers(sex);
}
```

Отлично, требуемый объект и его поведение мы описали. Теперь можно переходить к **JUnit**, но для начала покажу как примерно будет выглядеть простой тест если мы его будет делать в **main**.

```
public static void main(String[] args) {
    new User("Евгений", 35, Sex.MALE);
    new User("Марина", 34, Sex.FEMALE);
    new User("Алина", 7, Sex.FEMALE);


    System.out.println("Все пользователи:");
    User.getAllUsers().forEach(System.out::println);
    System.out.println("Все пользователи: MALE");
    User.getAllUsers(Sex.MALE).forEach(System.out::println);
    System.out.println("Все пользователи: FEMALE");
    User.getAllUsers(Sex.FEMALE).forEach(System.out::println);
    System.out.println("================================================");
    System.out.println("       всех пользователей: " + User.getHowManyUsers
    System.out.println("  всех пользователей MALE: " + User.getHowManyUsers
    System.out.println("всех пользователей FEMALE: " + User.getHowManyUsers
    System.out.println("================================================");
    System.out.println("       общий возраст всех пользователей: " + User.g
    System.out.println("  общий возраст всех пользователей MALE: " + User.g
    System.out.println("общий возраст всех пользователей FEMALE: " + User.g
    System.out.println("================================================");
    System.out.println("       средний возраст всех пользователей: " + User
    System.out.println("  средний возраст всех пользователей MALE: " + User
    System.out.println("средний возраст всех пользователей FEMALE: " + User
    System.out.println("================================================");
}
```

Вывод в консоль получим примерно такой, а дальше сравниваем получили ли мы нормальную работу. Можно конечно углубиться, написать логику сравнения, и посмотреть, что скажет наше вычисление, при том что мы не будем уверены, что все смогли предусмотреть.

```
//output
Все пользователи:
User{id=1, name='Евгений', age=35, sex=MALE}
User{id=2, name='Марина', age=34, sex=FEMALE}
User{id=3, name='Алина', age=7, sex=FEMALE}
Все пользователи: MALE
User{id=1, name='Евгений', age=35, sex=MALE}
Все пользователи: FEMALE
User{id=2, name='Марина', age=34, sex=FEMALE}
User{id=3, name='Алина', age=7, sex=FEMALE}
```

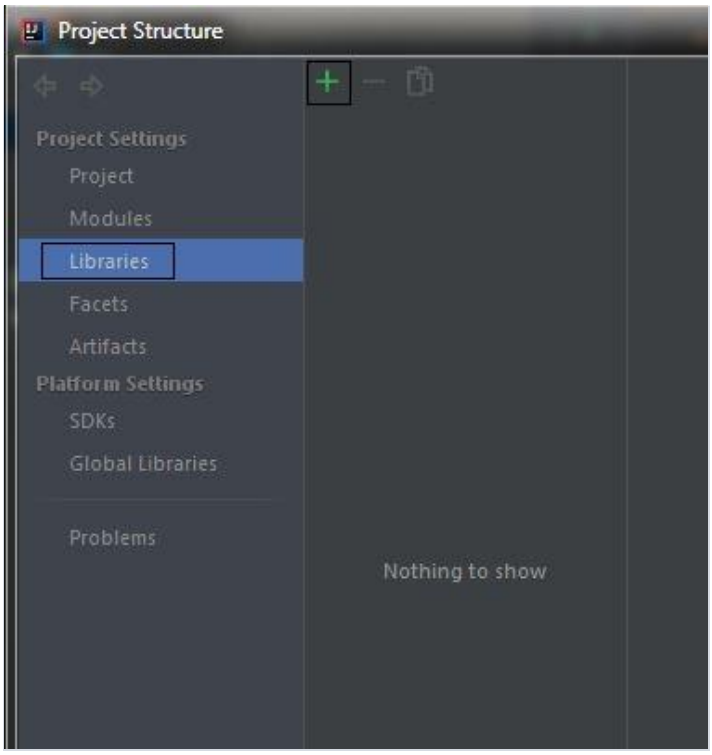
```
================================================
       всех пользователей: 3
  всех пользователей MALE: 1
всех пользователей FEMALE: 2
================================================
       общий возраст всех пользователей: 76
  общий возраст всех пользователей MALE: 35
общий возраст всех пользователей FEMALE: 41
================================================
       средний возраст всех пользователей: 25
  средний возраст всех пользователей MALE: 35
средний возраст всех пользователей FEMALE: 20
================================================

Process finished with exit code 0
```

Нас этот исход не устраивает, долой тесты **main**, нам нужен **JUnit**.

Как подключить JUnit к проекту

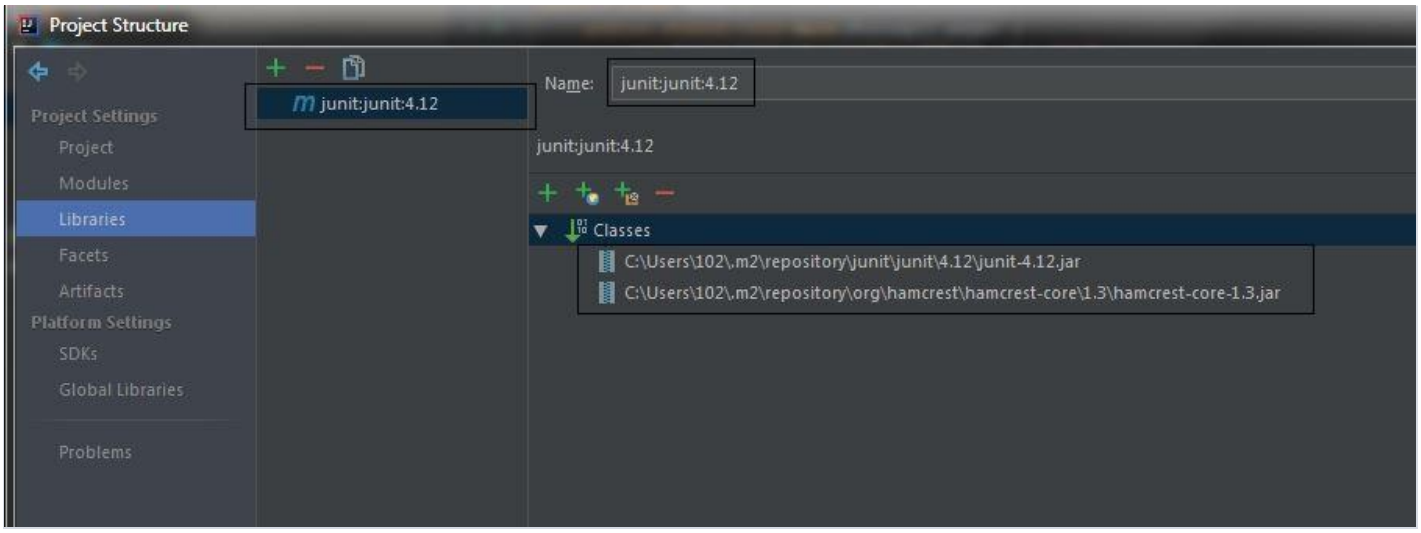
Возникает вопрос, как его подключить к проекту. Для знающих вариант с **Maven**брать не буду, так как это совсем другая история. ;)

Открываем структуру проекта Ctrl + Alt + Shift + S -> Libraries -> жмем + (New Project Library) -> выбираем from Maven



дальше видим такое окно, в строку поиска вводим “ junit:junit:4.12 ” ждем пока найдет -> OK! -> OK!

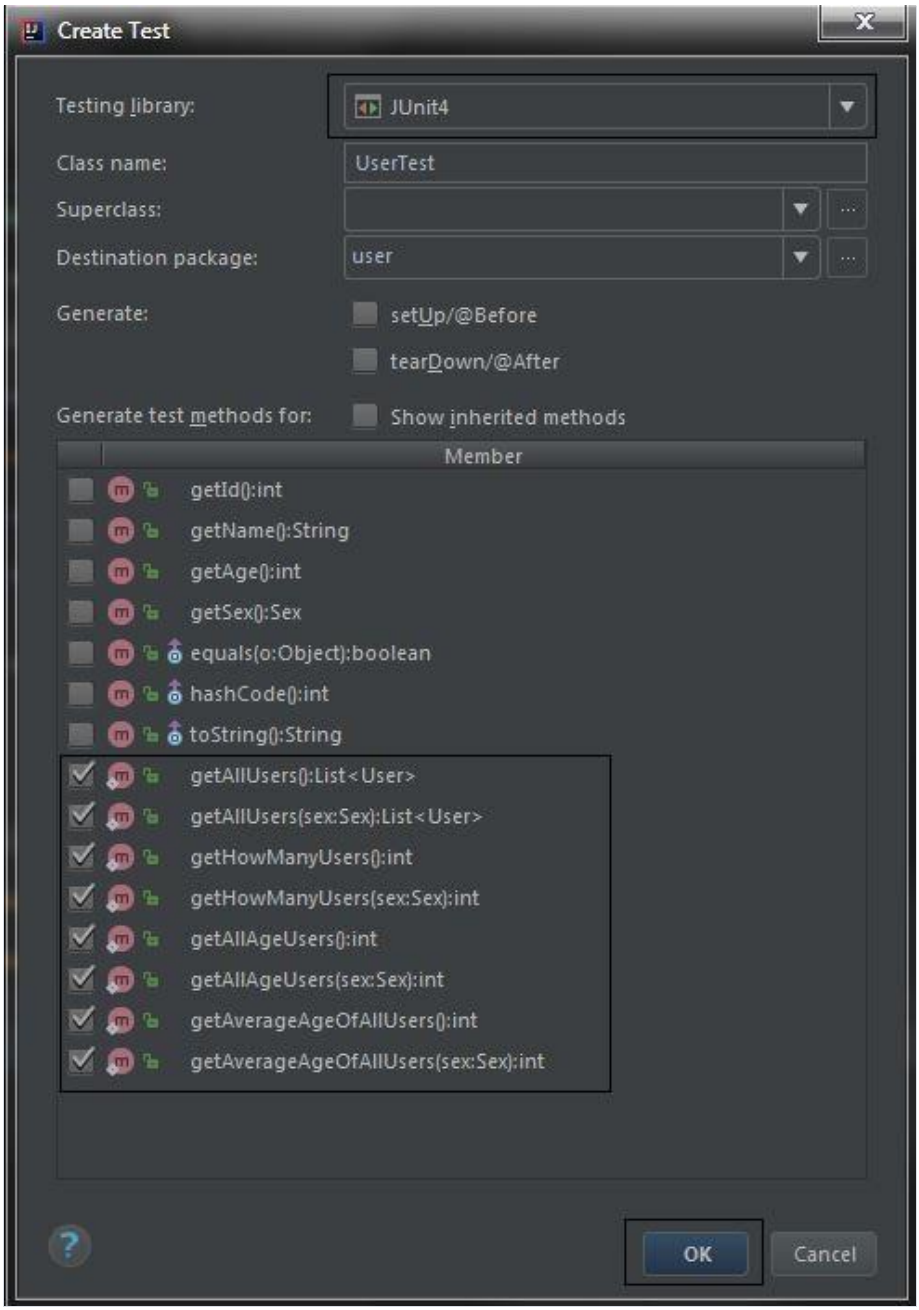
Должен получиться такой результат



Жмем OK, поздравлю JUnit добавлен к проекту. Едем дальше.

Теперь нам нужно создать тесты для нашего Java класса, ставим курсор на название класса `User` -> жмем Alt + Enter -> выбираем create Test.

Мы должны увидеть окно, в котором нам нужно выбрать библиотеку JUnit4 -> выбрать методы которые собираемся тестировать -> OK



Идея сама создаст класс `UserTest`, это и есть класс, в котором мы будем покрывать наш код тестами. Приступим:

Наш первый @Test

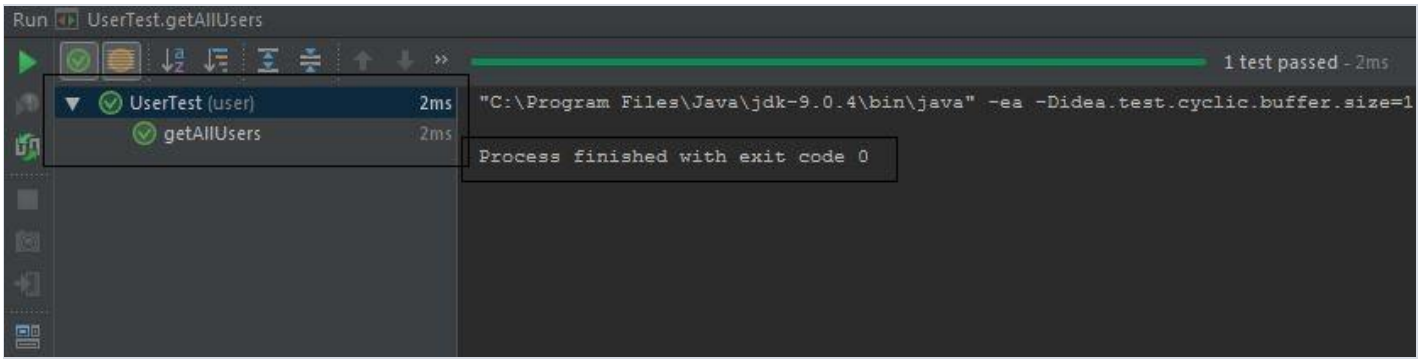
Создадим наш первый **@Test** метода `getAllUsers()` – это метод который должен вернуть всех пользователей. Тест будет выглядеть примерно так:

```
1  @Test
2  public void getAllUsers() {
3      //создаем тестовые данные
```

```
    User user = new User("Евгений", 35, Sex.MALE);
    User user1 = new User("Марина", 34, Sex.FEMALE);
    User user2 = new User("Алина", 7, Sex.FEMALE);

    //создаем список expected и заполняем его данными нашего метода
    List<User> expected = User.getAllUsers();

    //создаем список actual в него помещаем данные для сравнения
    //то что мы предпологиаем метод должен вернуть
    List<User> actual = new ArrayList<>();
    actual.add(user);
    actual.add(user1);
    actual.add(user2);

    //запускаем тест, в случае если список expected и actual не будут равны
    //тест будет провален, о результатах теста читаем в консоли
    Assert.assertEquals(expected, actual);
}
```

Тут мы создаем несколько тестовых пользователей -> создаем список `expected` в который поместим пользователей которых нам вернет метод `getAllUsers()` -> создадим список `actual` в который поместим пользователей которых мы предполагаем что метод `getAllUsers()` Assert.assertEquals(actual, expected) ему мы и передадим списки, инспектируемый и актуальный. Этот метод проверит объекты в предоставленных списках и выдаст результат теста. Метод будет сравнивать все поля объектов, даже пройдется по полям родителей, если есть наследование.

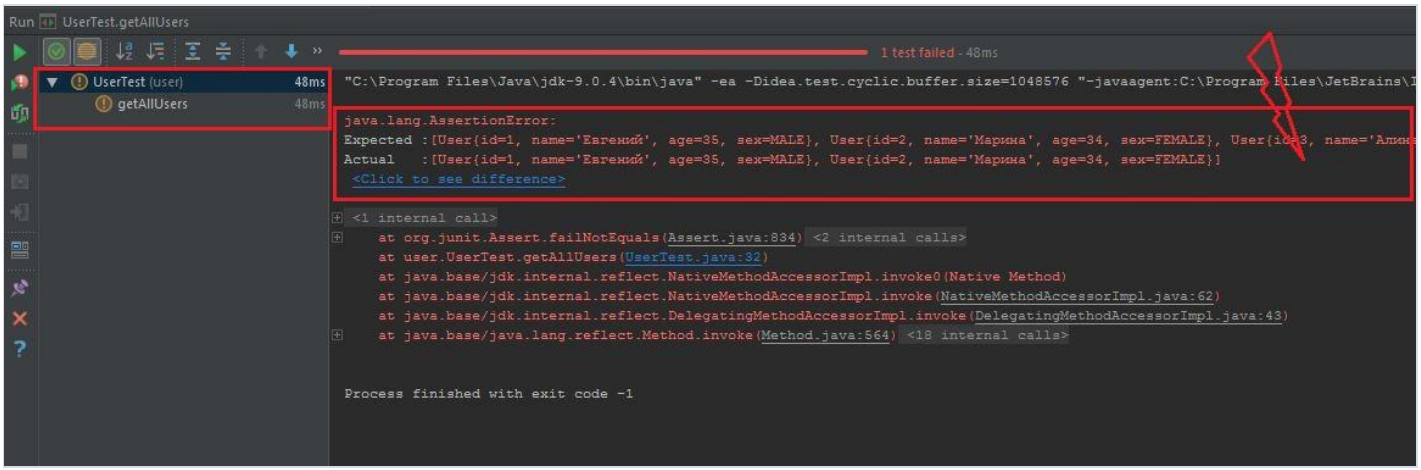
Запускаем первый тест...



Тест выполнен успешно. Теперь попробуем сделать так, чтобы тест был провален, для этого нам нужно изменить один из списков теста, сделаем это путем, закомментировав добавление одного пользователя в список `actual`,

```java
@Test
public void getAllUsers() {
    //создаем тестовые данные
    User user = new User("Евгений", 35, Sex.MALE);
    User user1 = new User("Марина", 34, Sex.FEMALE);
    User user2 = new User("Алина", 7, Sex.FEMALE);

    //создаем список expected и заполняем его данными нашего метода
    List<User> expected = User.getAllUsers();

    //создаем список actual в него помещаем данные для сравнения
    //то что мы предпологиаем метод должен вернуть
    List<User> actual = new ArrayList<>();
    actual.add(user);
    actual.add(user1);
    //actual.add(user2);

    //запускаем тест, в случае если список expected и actual не будут равны
    //тест будет провален, о результатах теста читаем в консоли
    Assert.assertEquals(expected, actual);
}
```

запускаем тест и видим следующее:

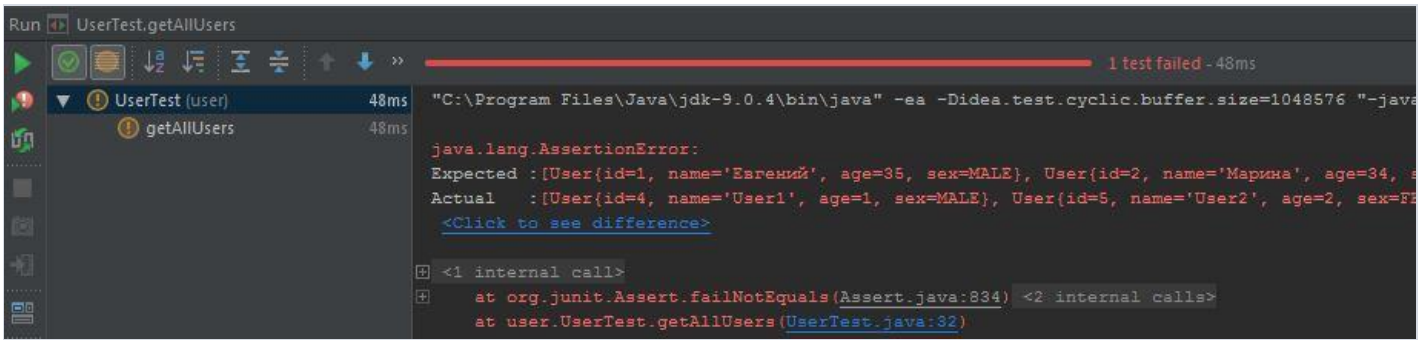


Теперь мы можем немного разобрать причину провала теста. Тут мы видим, что в инспектируемом списке больше пользователей чем в актуальном. Это и есть причина провала. А в main мы можем проверить так? JUnit : main = 1 : 0. Давайте посмотрим как будет выглядеть тест, если в нем будут полностью разные объекты, сделаем это так:

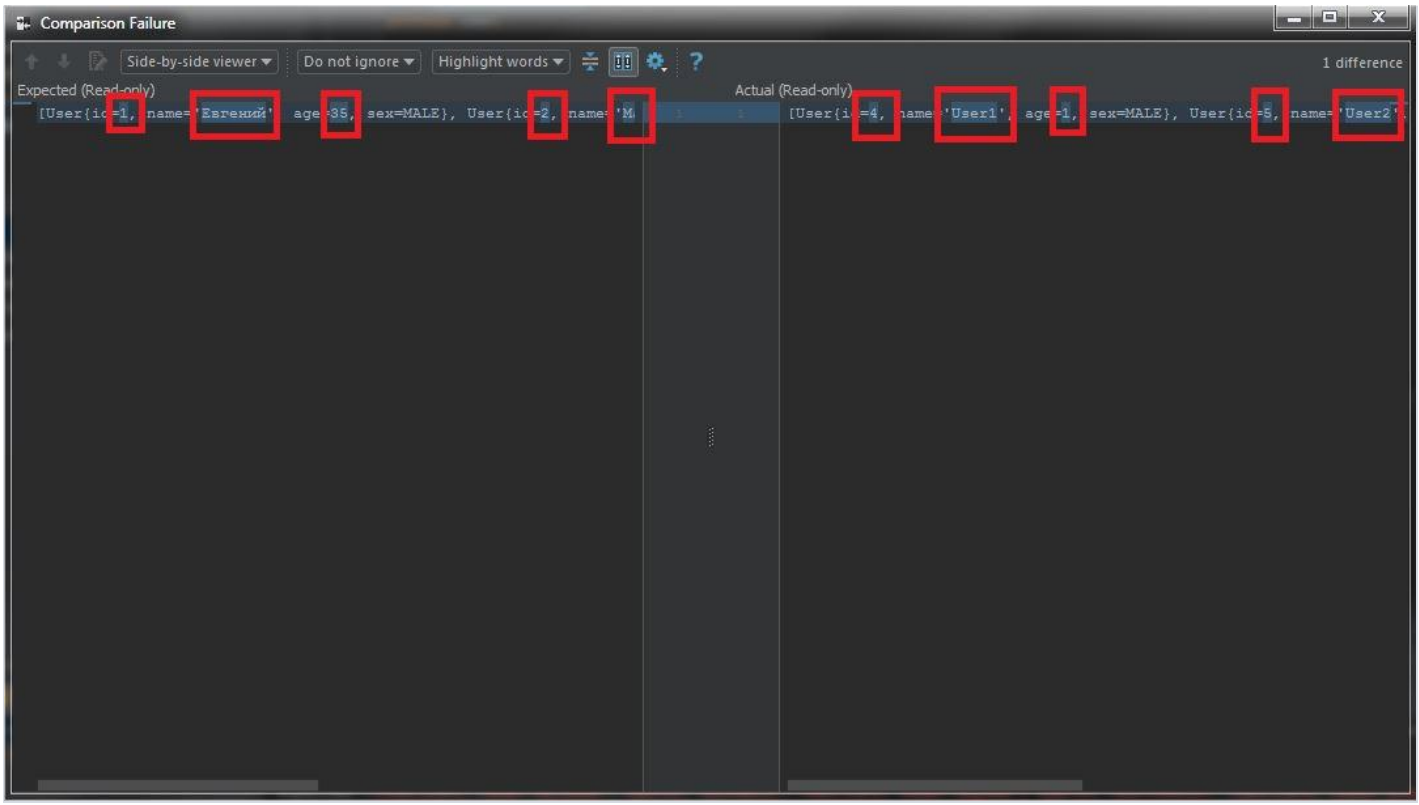
```java
@Test
public void getAllUsers() {
```

```
    //создаем тестовые данные
    User user = new User("Евгений", 35, Sex.MALE);
    User user1 = new User("Марина", 34, Sex.FEMALE);
    User user2 = new User("Алина", 7, Sex.FEMALE);

    //создаем список expected и заполняем его данными нашего метода
    List<User> expected = User.getAllUsers();

    //создаем список actual в него помещаем данные для сравнения
    //то что мы предпологиаем метод должен вернуть
    List<User> actual = new ArrayList<>();
    actual.add(new User("User1", 1, Sex.MALE));
    actual.add(new User("User2", 2, Sex.FEMALE));
    actual.add(new User("User3", 3, Sex.MALE));

    //запускаем тест, в случае если список expected и actual не будут равны
    //тест будет провален, о результатах теста читаем в консоли
    Assert.assertEquals(expected, actual);
}
```

вот что будет в консоли:



тут сразу видно что в сравниваемых списках разные пользователи, еще мы можем кликнуть на &ltClick to see difference> получим такое окно, где можно посмотреть подробно с какими данными у нас проблема. **IDEA** подсветит все поля в которых есть различия.



main такое может? - нет. JUnit : main = 2 : 0

Ну что, пойдем дальше у нас еще куча методов, которые нужно покрыть тестами), но подождите, а ведь будет не плохо, проверить, а не будет ли нам метод `getAllUsers()` возвращать `null`, ведь примерно так нас на задачах **JavaRush** ловит валидатор). Сделаем это, делов то на три копейки …

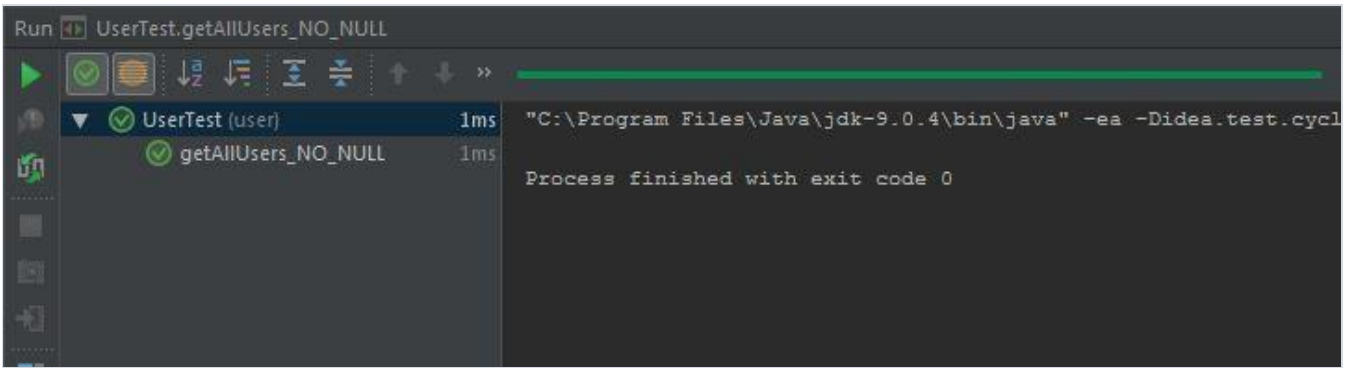
```
@Test
public void getAllUsers_NO_NULL() {
    //добавим проверку на null
    List<User> expected = User.getAllUsers();
    Assert.assertNotNull(expected);
}
```

Да, да примерно так валидатор ловит наш ~~говно~~ код на `null` ;) Теперь запустим этот тест, и посмотрим, что он нам покажет. А покажет он ошибку, как ???? как же тут можно было допустить ошибку теста)))

И тут мы можем пожинать первые плоды покрытия своего кода тестами. Как вы помните, поле `allUsers` мы инициализировали в конструкторе, и значит при вызове метода `getAllUsers()`, мы обратимся к объекту, который еще не был инициализирован. Будем править, уберем инициализацию из конструктора, и сделаем ее при объявлении поля.

```
1  private static Map<Integer, User> allUsers = new HashMap<>();
2
3      public User(String name, int age, Sex sex) {
4          this.name = name;
5          this.age = age;
6          this.sex = sex;
7
8          if (!hasUser()) {
9              countId++;
10             this.id = countId;
11             allUsers.put(id, this);
12         }
13     }
```

Запустим тест, теперь все хорошо.



не думаю что в main легко будет отловить NPE, думаю вы согласитесь что счет JUnit : main = 3 : 0

Дальше я все методы покрою тестами, и дам вам посмотреть, как это будет выглядеть...

Теперь класс тестов у нас выглядит так:

```
package user;

import org.junit.Assert;
import org.junit.Test;

import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.*;

public class UserTest {

    @Test
    public void getAllUsers() {
        //создаем тестовые данные
        User user = new User("Евгений", 35, Sex.MALE);
        User user1 = new User("Марина", 34, Sex.FEMALE);
        User user2 = new User("Алина", 7, Sex.FEMALE);

        //создаем список expected и заполняем его данными нашего метода
        List<User> expected = User.getAllUsers();

        //создаем список actual в него помещаем данные для сравнения
        //то что мы предпологиаем метод должен вернуть
        List<User> actual = new ArrayList<>();
        actual.add(user);
        actual.add(user1);
        actual.add(user2);

        //запускаем тест, в случае если список expected и actual не будут равн
        //тест будет провален, о результатах теста читаем в консоли
        Assert.assertEquals(expected, actual);
    }

    @Test
    public void getAllUsers_NO_NULL() {
        //добавим проверку на null
        List<User> expected = User.getAllUsers();
        Assert.assertNotNull(expected);
    }

    @Test
```

```
43      public void getAllUsers_MALE() {
44          User user = new User("Евгений", 35, Sex.MALE);
45          User user1 = new User("Марина", 34, Sex.FEMALE);
46          User user2 = new User("Алина", 7, Sex.FEMALE);
47
48          List<User> expected = User.getAllUsers(Sex.MALE);
49
50          List<User> actual = new ArrayList<>();
51          actual.add(user);
52
53          Assert.assertEquals(expected, actual);
54      }
55
56      @Test
57      public void getAllUsers_MALE_NO_NULL() {
58          //добавим проверку на null
59          List<User> expected = User.getAllUsers(Sex.MALE);
60          Assert.assertNotNull(expected);
61      }
62
63      @Test
64      public void getAllUsers_FEMALE() {
65          User user = new User("Евгений", 35, Sex.MALE);
66          User user1 = new User("Марина", 34, Sex.FEMALE);
67          User user2 = new User("Алина", 7, Sex.FEMALE);
68
69          List<User> expected = User.getAllUsers(Sex.FEMALE);
70
71          List<User> actual = new ArrayList<>();
72          actual.add(user1);
73          actual.add(user2);
74
75          Assert.assertEquals(expected, actual);
76      }
77
78      @Test
79      public void getAllUsers_FEMALE_NO_NULL() {
80          //добавим проверку на null
81          List<User> expected = User.getAllUsers(Sex.FEMALE);
82          Assert.assertNotNull(expected);
83      }
84
85      @Test
86      public void getHowManyUsers() {
87          User user = new User("Евгений", 35, Sex.MALE);
88          User user1 = new User("Марина", 34, Sex.FEMALE);
89          User user2 = new User("Алина", 7, Sex.FEMALE);
90
```

```
91          int expected = User.getHowManyUsers();
92
93          int actual = 3;
94
95          Assert.assertEquals(expected, actual);
96      }
97
98      @Test
99      public void getHowManyUsers_MALE() {
100         User user = new User("Евгений", 35, Sex.MALE);
101         User user1 = new User("Марина", 34, Sex.FEMALE);
102         User user2 = new User("Алина", 7, Sex.FEMALE);
103
104         int expected = User.getHowManyUsers(Sex.MALE);
105
106         int actual = 1;
107
108         Assert.assertEquals(expected, actual);
109     }
110
111     @Test
112     public void getHowManyUsers_FEMALE() {
113         User user = new User("Евгений", 35, Sex.MALE);
114         User user1 = new User("Марина", 34, Sex.FEMALE);
115         User user2 = new User("Алина", 7, Sex.FEMALE);
116
117         int expected = User.getHowManyUsers(Sex.FEMALE);
118
119         int actual = 2;
120
121         Assert.assertEquals(expected, actual);
122     }
123
124     @Test
125     public void getAllAgeUsers() {
126         User user = new User("Евгений", 35, Sex.MALE);
127         User user1 = new User("Марина", 34, Sex.FEMALE);
128         User user2 = new User("Алина", 7, Sex.FEMALE);
129
130         int expected = User.getAllAgeUsers();
131
132         int actual = 35 + 34 + 7;
133
134         Assert.assertEquals(expected, actual);
135     }
136
137     @Test
138     public void getAllAgeUsers_MALE() {
```

```
        User user = new User("Евгений", 35, Sex.MALE);
        User user1 = new User("Марина", 34, Sex.FEMALE);
        User user2 = new User("Алина", 7, Sex.FEMALE);

        int expected = User.getAllAgeUsers(Sex.MALE);

        int actual = 35;

        Assert.assertEquals(expected, actual);
    }

    @Test
    public void getAllAgeUsers_FEMALE() {
        User user = new User("Евгений", 35, Sex.MALE);
        User user1 = new User("Марина", 34, Sex.FEMALE);
        User user2 = new User("Алина", 7, Sex.FEMALE);

        int expected = User.getAllAgeUsers(Sex.FEMALE);

        int actual = 34 + 7;

        Assert.assertEquals(expected, actual);
    }
}
```

Да не маленький получился, а что же будет при работе с большими проектами. Что же тут можно сократить, оценив все можно заметить, что тестовые данные мы создаем в каждом тесте, и тут нам на помощь приходят аннотации. Возьмем **@Before** - Аннотация **@Before** указывает на то, что метод будет выполнятся перед каждым тестируемым методом **@Test**.

Вот так теперь будет выглядеть наш класс тестов с аннотацией **@Before**:

```
package user;

import org.junit.Assert;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import java.util.ArrayList;
import java.util.List;

import static org.junit.Assert.*;

public class UserTest {
    private User user;
```

```
15      private User user1;
16      private User user2;
17
18      @Before
19      public void setUp() throws Exception {
20          user = new User("Евгений", 35, Sex.MALE);
21          user1 = new User("Марина", 34, Sex.FEMALE);
22          user2 = new User("Алина", 7, Sex.FEMALE);
23      }
24
25      @Test
26      public void getAllUsers() {
27          List<User> expected = User.getAllUsers();
28
29          List<User> actual = new ArrayList<>();
30          actual.add(user);
31          actual.add(user1);
32          actual.add(user2);
33
34          Assert.assertEquals(expected, actual);
35      }
36
37      @Test
38      public void getAllUsers_NO_NULL() {
39          List<User> expected = User.getAllUsers();
40          Assert.assertNotNull(expected);
41      }
42
43      @Test
44      public void getAllUsers_MALE() {
45          List<User> expected = User.getAllUsers(Sex.MALE);
46
47          List<User> actual = new ArrayList<>();
48          actual.add(user);
49
50          Assert.assertEquals(expected, actual);
51      }
52
53      @Test
54      public void getAllUsers_MALE_NO_NULL() {
55          //добавим проверку на null
56          List<User> expected = User.getAllUsers(Sex.MALE);
57          Assert.assertNotNull(expected);
58      }
59
60      @Test
61      public void getAllUsers_FEMALE() {
62          List<User> expected = User.getAllUsers(Sex.FEMALE);
```