

ЛЕКЦИЯ I © 2020 Нет Ит

JAVA OOP: Юнит тестове

Теодор Костадинов



SOFTWARE
ACADEMY



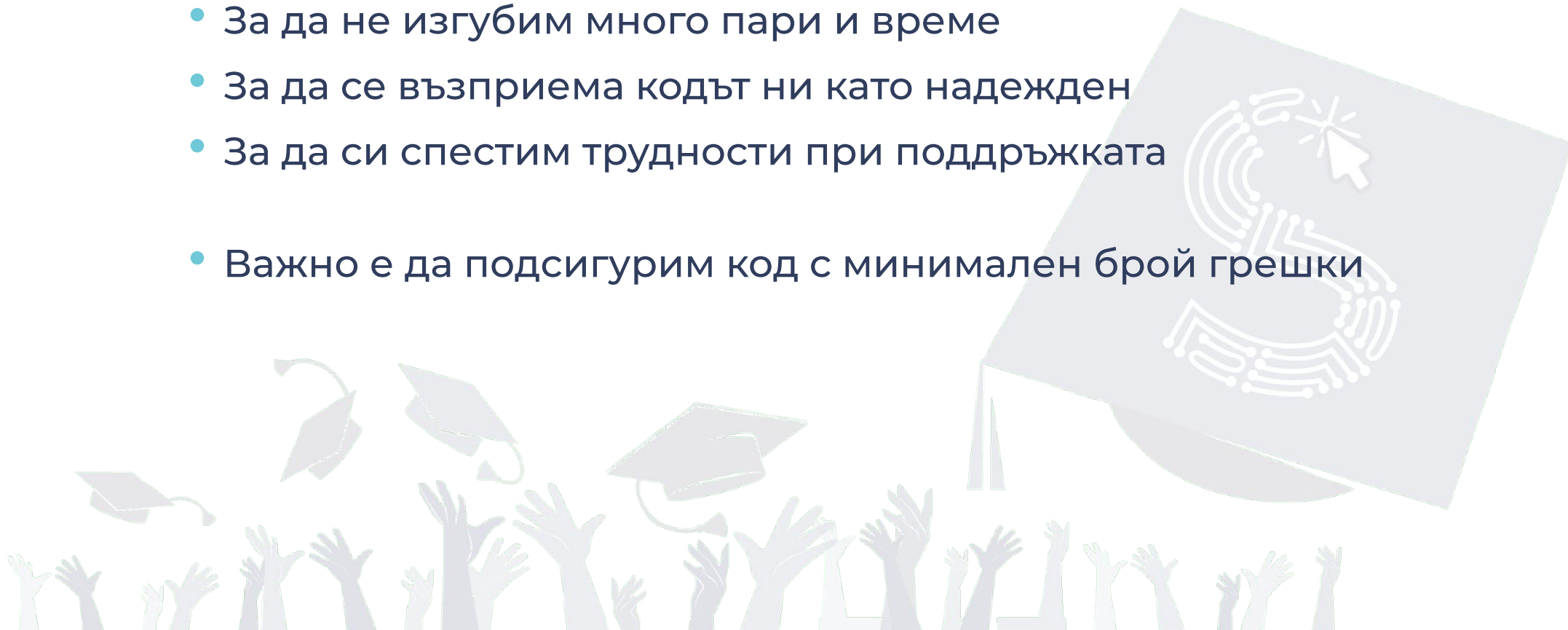
Съдържание

1. Какво са тестове
2. Защо и кога трябва да пишем тестове
3. Дефиниции, свързани с тестовете
4. Какво е Unit Test и как се пише
5. Какво е JUnit и за какво се използва
6. Добри практики за писане и именуване на тестове
7. Класът Assert



Защо трябва да тестваме софтуера?

- За да не произвеждаме код с бъгове
- За да не изгубим много пари и време
- За да се възприема кодът ни като надежден
- За да си спестим трудности при поддръжката
- Важно е да подсигурием код с минимален брой грешки



Какво са тестове?

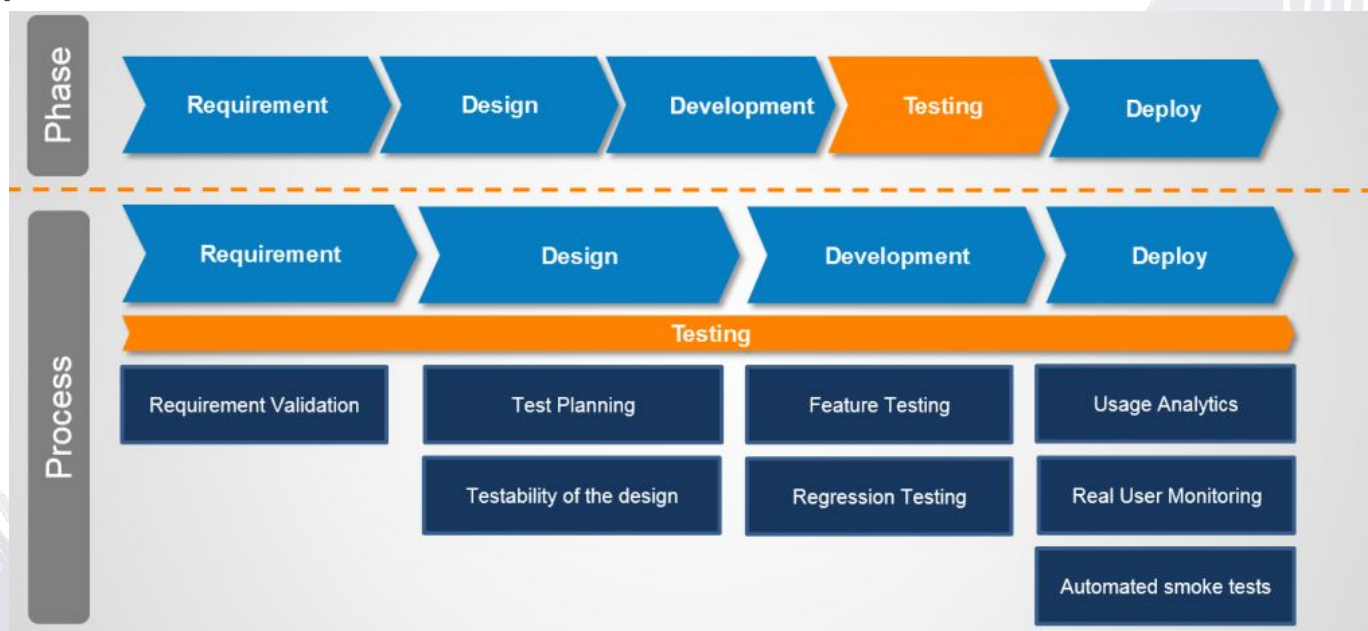
- Последователност от действия, която подsigурява правилното изпълнение на дадена функционалност
- Тестовите се създават по определен тестов случай
- Сравнение на очакваният и реалният резултат от тестваното парче код

```
@Test  
public void testLoginWhenPasswordIsIncorrectThenReturnFalse(){  
    //test Login(String password) method here  
}
```



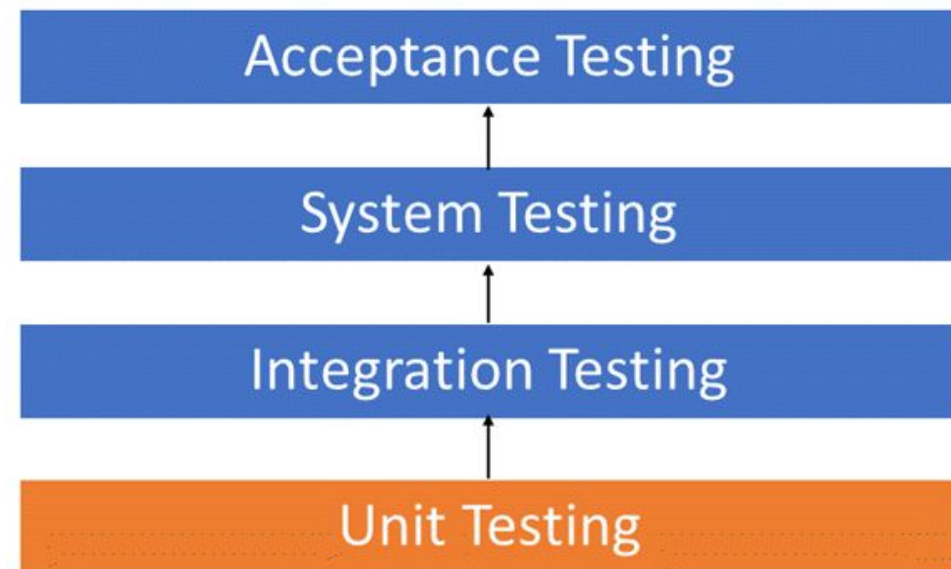
Кога трябва да пишем тестове?

- Тестовите не са етап от създаването на програма, те са процес
- Тестове трябва да се пишат успоредно с останалият код
- Не е добра практика всички тестове да се напишат накрая
- Ако пишем тестовите си навреме, ще си спестим много проблеми в бъдещето



Видове автоматични тестове

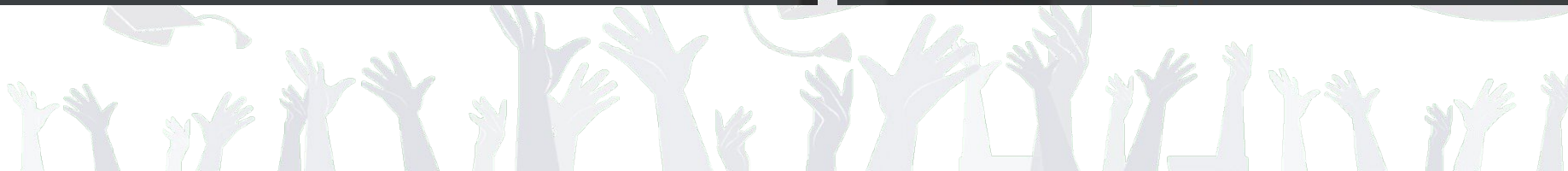
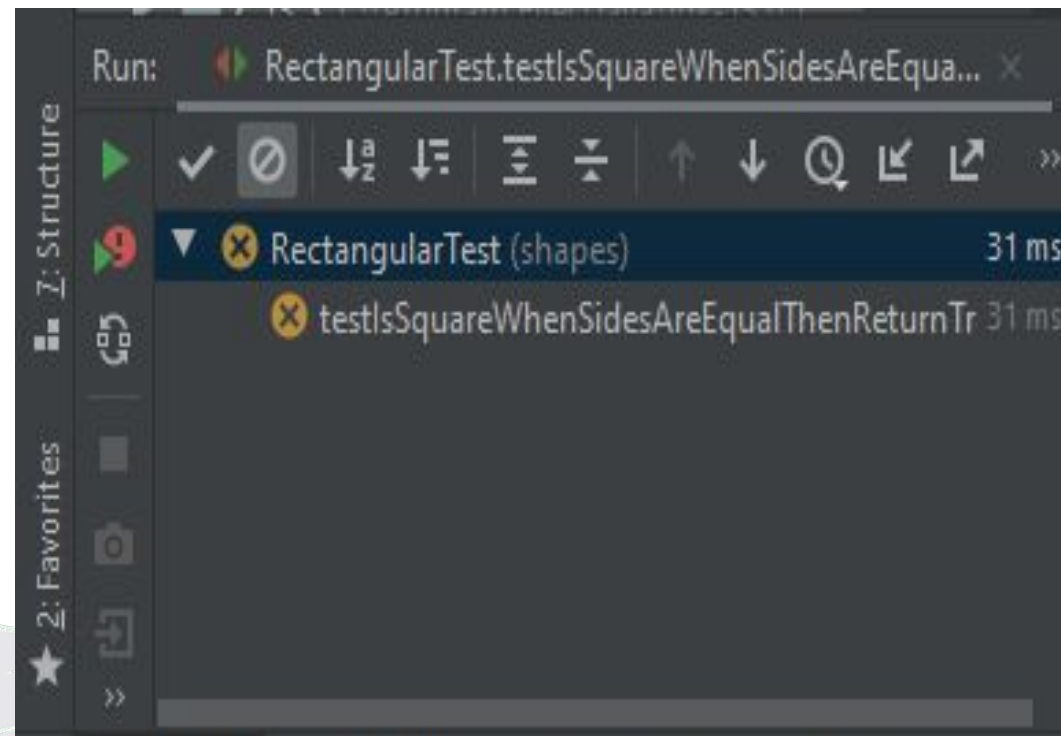
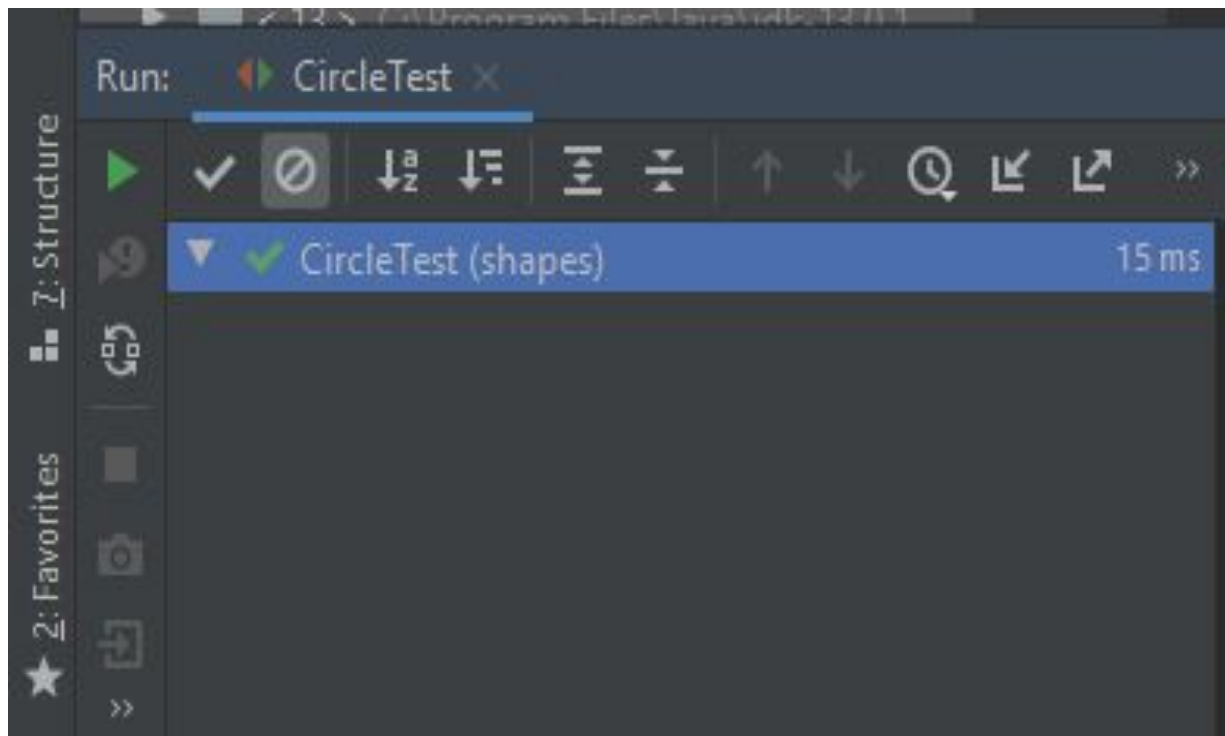
- Функционални тестове
 - Unit
 - integration
- Нефункционални тестове
 - Performance
 - stress
 - crash
 - security
 - usability



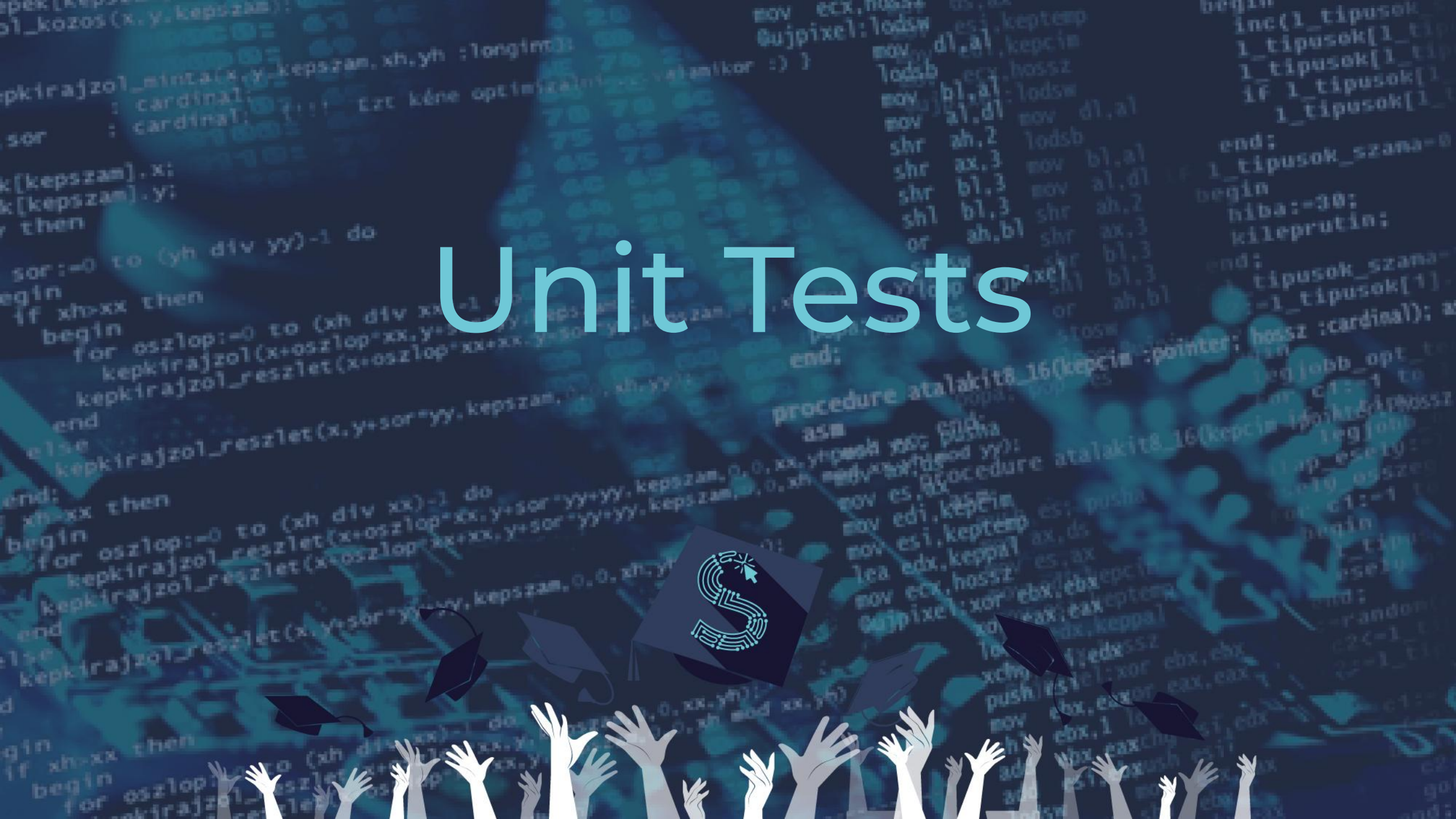
Дефиниции

- Продуктивен код (code under test) - кодът, който реализира потребителските изисквания
- Test coverage или code coverage - процентът на продуктивния код, който се тества от автоматични тестове
- Test driven development (TDD) е методология, при която тестовете се пишат преди продуктивния код
 - Подсигуряваме, че когато тестът мине, кодът, който сме написали работи по начина, по който искаме
 - Не се изкушаваме да нагласим теста, така че да минава
 - щом даден тест бъде удовлетворен (т.е. минава успешно, стане „зелен“), съответният use-case е реализиран („done“)

Какво означава тест да “мине” или “фейлне”?

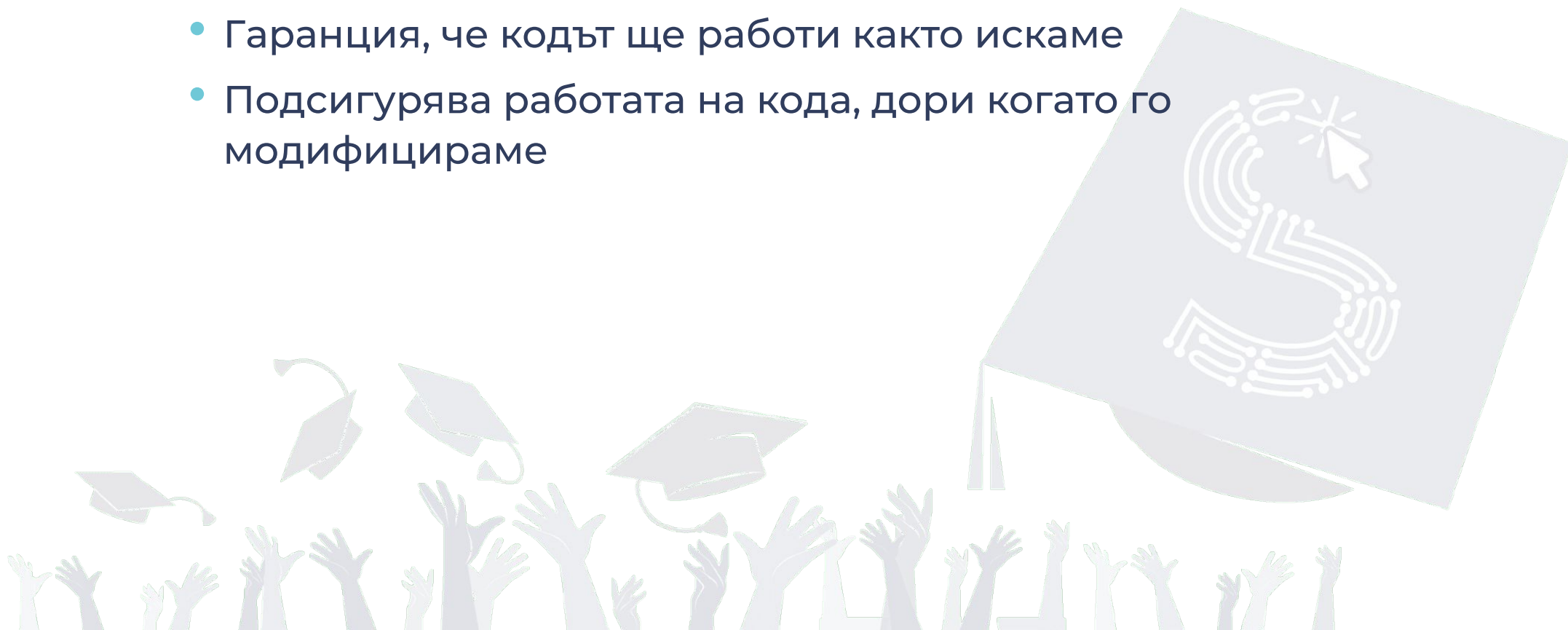


Unit Tests



Unit tests

- Код, който изпълнява специфична, „атомарна“ функционалност на кода, която да бъде тествана
- Тества се малък фрагмент код (метод или най-много клас)
- Гаранция, че кодът ще работи както искаме
- Подсигурява работата на кода, дори когато го модифицираме



Анотации

- @Test
 - обозначава метод като тестов метод
- @Test(timeout = 100)
 - фейлва, ако изпълнението на метода продължи повече от 100 милисекунди



JUnit

- Най-популярният framework за писане на тестове в Java
- Базира се на анотации
- Всеки тест е метод с анотация @Test
- Тестовите се намират в клас, който се използва само за тестване (test case)



Добри практики

- Тривиален код като гетъри и сетъри няма нужда от тестове
- Тестовете трябва да са кратки, точни и ясни
- 70-80% code coverage
- Един тест трябва да покрива само един сценарий
- Тестовете трябва да са независими
- Тестовете са безполезни, ако кодът ни не е добре разделен на методи!
- Името на тестовия клас = Името на тествания клас + Test
пр: Login и LoginTest
- Имената на тестовете започват с test, след това следва името на тествания метод и кратко описание на тестовия сценарий
`test<methodName>When<scenario>Then<expectedResult>`
пр: `testLoginWhenPasswordIsIncorrectThenReturnFalse`

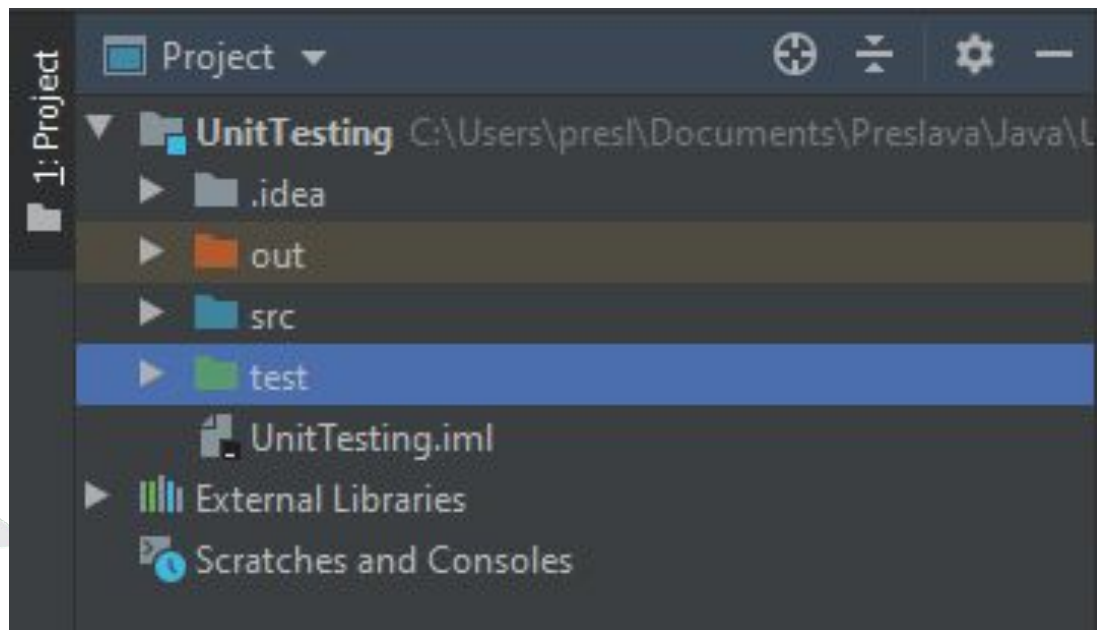
Ред на изпълнение

- По default , тестовете се изпълняват в случаен ред
- Добре написаните тестове са независими и би трябвало да работят без значение от реда им на изпълнение
- От JUnit 4.11 натам тестовете може да се подредят, но не е препоръчително!



Къде “живеят” тестовете

- Обикновено unit тестовете се разполагат в отделен проект или в отделна source директория, за да са отделени от продуктивния код



Assert



Използване на assert

- Класът Assert съдържа статични методи, служещи за проверяване на очакван резултат.

```
public class HelloWorld{  
    public String sayHello(){  
        return "Hello world!";  
    }  
}
```

```
public class HelloWorldTest {  
  
    @Test  
    public void testSayHello() {  
        HelloWorld hello = new HelloWorld();  
        Assert.assertEquals("Method doesn't return  
expected value", "Hello world!",  
hello.sayHello());  
    }  
}
```

Използване на assert

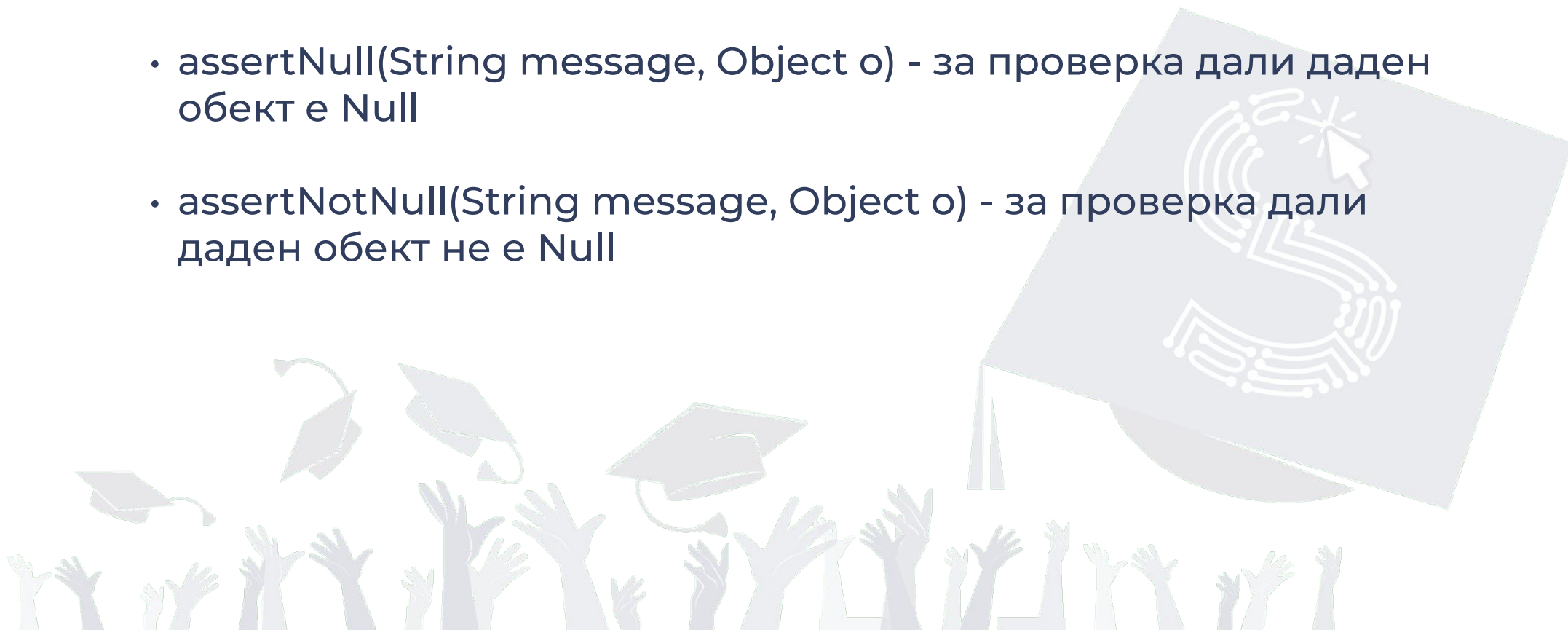
- `assertArrayEquals(String message, expected, actual)` - за сравнение на масиви по дължина и съдържание
- `assertEquals(String message, expected, actual, delta)` - за сравнение на числа с плаваща запетая. Delta определя точността на сравнението
- `assertTrue(String message, boolean condition)`

```
public class Calculator {  
    public boolean areEqual(int a, int b){  
        return a == b;  
    }  
}
```

```
public class CalculatorTest {  
    private Calculator calc = new Calculator();  
  
    //in this case test passes  
    @Test  
    public void testAreEqualWhenInputNumbersAreSame(){  
        Assert.assertTrue("Method doesn't  
            return expected value", calc.areEqual(1, 1));  
    }  
  
    //in this case test fails  
    @Test  
    public void testAreEqualWhenInputNumbersAreDifferent(){  
        Assert.assertTrue("Method doesn't  
            return expected value", calc.areEqual(1, 2));  
    }  
}
```

Използване на assert

- `assertFalse(String message, boolean condition)` - за проверка дали дадено условие е лъжа
- `assertNull(String message, Object o)` - за проверка дали даден обект е Null
- `assertNotNull(String message, Object o)` - за проверка дали даден обект не е Null



@Before & @After

- Когато аотираме метод по този начин, този метод ще се извика преди/след всеки тест

```
public class NumberListTest {  
  
    private NumberList numList;  
  
    @Before  
    public void initialize() {  
        numList = new NumberList();  
    }  
  
    @After  
    public void initialize() {  
        numList.clear();  
    }  
  
    @Test  
    public void testFindSizeWhenNoElementsThenReturnZero() {  
        //test here  
    }  
}
```

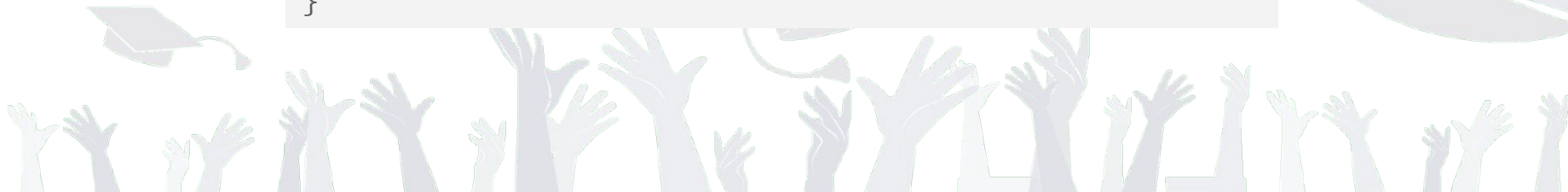


@BeforeClass & @AfterClass

- Когато аотираме метод по този начин, този метод ще се извика преди първият/след последният тест

```
@BeforeClass
public static void setup() {
    LOG.info("startup - creating DB connection");
}

@AfterClass
public static void tearDown() {
    LOG.info("closing DB connection");
}
```



ВЪПРОСИ?



Почивка

до 20:15



Резюме



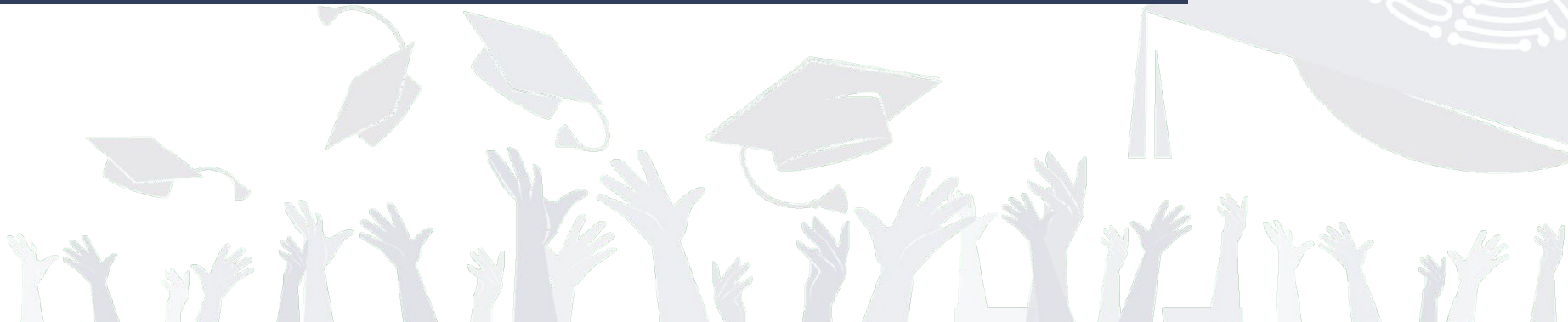
- Тестовите са важни!
- Винаги трябва да пишем тестове
- Тестовите гарантират качеството на кода
- Unit тестовите трябва да са точни и ясни
- Как да пишем и именуваме тестове
- За да има полза от тестовите, кодът ни трябва да е добре разделен на методи



Ресурси



- [Docs](#)
- [GitHub Repo with Demos](#)
-



Задачи за упражняване



Задача 1

Създадена е програма, която ни помага да разбираме различните фигури по-добре. Тя намира обиколките и лицата им, както и някои специфични за фигурата особености. Имплементирани са класове Circle, Rectangular и Triangle.

Кода може да откриете тук:

<https://github.com/NetIt-Android-Course/java-oop-2020/tree/master/22-unit-tests-part-two/src/shapes>



Ами сега?

Задача за упражнение

Задача 1

- Нека да разгледаме и да коментираме:
 - имплементацията на кода
 - вече написаните тестове за класовете Circle и Rectangular.
- По подобен начин нека сега разгледаме логиката на класа Triangle и да напишем подходящи тестове, с които да тестваме поведението на кода. При нужда да се поправят грешките.



Ами сега?

Задача за упражнение

Задача 2

- Създадено е приложение, което цели събирането на такса за магистрала. На магистралата може да има два типа превозни средства - Cars и Motorbikes. Всеки от двата класа има член-данни и имплементирани методи. Програмата пази данните за всички превозни средства, които се движат по пътя в момента. Превозно средство може да се качва или да слиза от магистралата.



Ами сега?

Задача за упражнение

Задача 2

- В тази задача ще влезем в ролята на QA. Това, което те правят ежедневно, е да се грижат, че кодът, написан от програмистите, работи правилно. Тестовете, които те пишат, трябва да покриват колкото може по-голям процент от сценариите, които могат да се разиграят, и да поправят откритите бъгове.
- Код може да откриете на <https://github.com/NetIt-Android-Course/java-oop-2020/tree/master/22-unit-tests-part-two/src/highway>



Ами сега?

Задача за упражнение

Задача 2

- За целта нека първо заедно да разгледаме следния код и да коментираме какви тестове могат да се напишат, за да се тества функционалността на готовото приложение.
- Както видяхме, понякога даже и програмистите допускат пропуски/грешки, които в някои случаи могат да се окажат фатални. Затова е нужно да се пишат достатъчно много тестове, които да откриват възможни проблемите. Нека сега заедно да поправим откритите проблеми.



Ами сега?

Задача за упражнение



Това домашно влиза в крайната ви оценка!

Домашно

Качвайте домашното си в ГитХъб и
слагайте линка тук:

<https://forms.gle/AcvCptCbSDizr2Ay6>



Задача 1



Довършете писането на тестове за задачата за фигурите. Тествайте напълно класовете Rectangular и Triangle. Тествайте всички възможни случаи, за които се сетите, и подсигурете, че кода работи правилно. Например методът isIsosceles(), който проверява дали даден триъгълник е равнобедрен, трябва да се тества поне три пъти - всяка страна с всяка друга



Задача 2



Довършете писането на тестове за задачата за магистралата. Разгледахме защо родителски клас `VehicleBase` е по-удачен от интерфейса `Vehicle`. Рефакторирайте класа `Motorbike` по същия начин както направихме с `Car`. Помислете как да си подсигурите, че няма как едно и също превозно средство да се движи два пъти на магистралата. Например проверете за наличие на такъв `registrationNumber` в листа.



!!!



Забележка: не е нужно да добавяте нова функционалност за тези задачи, тествайте само вече имплементираната, но ако решите да го направите не забравяйте прилежащите тестове. При наличие на грешка в логиката или структурата (разделяне по класове, наследяване и т.н.) на приложението - поправете я.



Задача 3

Подобно на последната задача, която направихме заедно, напишете подходящи тестове за предоставения код.

- Условието може да намерите на следващия слайд
- Имплементация:
<https://github.com/NetIt-Android-Course/java-oop-2020/tree/master/22-unit-tests-part-two/src/cart>

и поправете грешките. Имайте предвид, че е възможно те да не са само в логиката на приложението, а може и някои добри практики да са нарушени.



Задача 3

Shopping cart 1/2

Да се имплементира пазарска количка, в която се съхраняват различни продукти, които искаме да купим от даден магазин.

- Количката да съхранява елементите си в Map, където ключът ви е съответния артикул, а стойността е количеството.
- Артикулите, които могат да бъдат добавяни в количката, са Apple и Chocolate. Може да имаме различни видове ябълки и шоколади.
- Всеки артикул се характеризира с име, описание и цена.



Задача 3



Shopping cart 2/2

Забележка 1: Два артикула са еднакви, ако всичките им характеристики са еднакви.
Забележка 2: Ако артикул, който не е наличен в количката, се добави за първи път, то стойността ще е 1. Ако обаче се опитаме да добавим същия продукт за втори път, вместо да създадем нов ключ, то просто увеличаваме наличната стойност с 1. Това става като се предефинира съответния @equals, който всеки обект има.

За да разберете напълно как работят map-овете, прочетете следните материали:

1. Как работят HashMap-овете в Java:

<https://medium.com/@om.prerna/lets-know-all-about-the-hashmap-in-java-5be57d2a986>

2. HashMaps quick guide: https://www.w3schools.com/java/java_hashmap.asp

3. Разликата между equals == и hascode:

<https://www.javaworld.com/article/3305792/comparing-java-objects-with-equals-and-hashcode.html>

4. Как се override-ват методите equals и hashCode на всеки обект:

<https://mkyong.com/java/java-how-to-overrides-equals-and-hashcode/>

Препоръчваме ви да използвате втория метод

© 2020 Нет Ит

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!



SOFTWARE
ACADEMY

