Testes Unitarios JUnit + Mockito

1. JUnit

▼ **Notations de teste**

1. @Before: Método executa antes de cada teste
2. @After Método executa depois de cada teste
3. @BeforeClass Método executa uma vez antes do inicio do primeiro teste
4. @AfterClass Metodo executa uma vez apos o ultimo teste

▼ **Asserts**

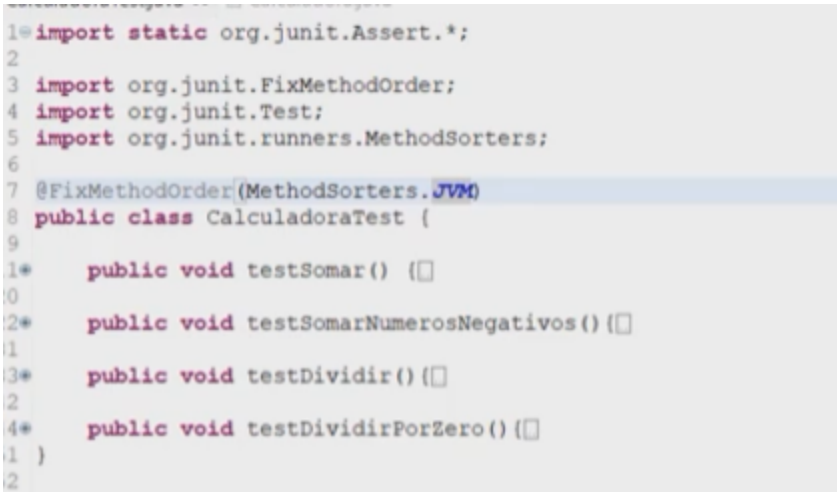
1. assertTrue() Verifica uma condição
2. assertEquals() compara um valor esperar com o atual
3. assertNotEquals() Verifica se o valor expectado e diferento do atual
4. assertSame() verifica se o valor expectado aponta pro mesmo endereço de memoria do atual
5. assertNotSame() verifica se o valor expectado não aponta pro mesmo endereço de memoria do atual
6. assertArrayEquals() verifica se o conteudo de um array e igual ao outro
7. assertNull() verifica se o valor passado é nulo
8. assertNotNull() verifica se o valor passado não é nulo

▼ **Expected**

Também pode ser colocado o erro esperado na annotation @TEST

▼ **ordenação de execução**

usando notation @FixMethodOrder()



▼ **Teste Parametrizados**

Usando a notation @RunWhit(Parameterized.class)

conseguimos criar teste parametrizadas criando assim varios testes sem precisar criar varios metodos de teste para testa a mesma função para cada parametro passar o Junit rodar novamente o teste com os parametros seguintes

```java
import java.util.Arrays;
import java.util.Collection;

@RunWith(Parameterized.class)
public class RetanguloTeste {

    private Integer base;
    private Integer altura;
    private Integer area;
    private Integer perimetro;

    @Parameterized.Parameters
    public static Collection<Object[]> parametros () {
        return Arrays.asList(new Object[][]{
                {20,10,200,60},
                {30,20,600,100}
        });
    }


    public RetanguloTeste(Integer base ,Integer altura, Integer area, Integer perimetro){
        this.base = base;
        this.altura = altura;
        this.area = area;
        this.perimetro = perimetro;
    }

    @Test
    public void testeCalculoDeArea() {
        Retangulo retangulo = new Retangulo(base,altura);
        Assert.assertEquals(area, retangulo.calcularArea());
    }

    @Test
```

▼ Bateria de testes

Com o passar do tempo nosso programa vai se enchendo de teste é fica demorado rodar todas as classes de testes de vez com suite conseguimos rodar todos os teste de uma so vez

```java
package com.company;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({CalculadoraTestes.class, PilhaLivrosTeste.class, RetanguloTeste.class})
public class BateriaDeTeste {
}
```

▼ Runners

Classes runners servem para rodar teste sem o auxilio da IDE

```
mport org.junit.runner.JUnitCore;
mport org.junit.runner.Result;
mport org.junit.runner.notification.Failure;

ublic class PilhaTestRunner {

   public static void main(String[] args) {
       Result resultado = JUnitCore.runClasses(LivroTest.class);

       List<Failure> falha = resultado.getFailures();

       for (Failure failure : falha) {
           System.out.println(failure.getMessage());
       }

       System.out.println("Resultado dos testes:" + resultado.wasSuccessful());

   }
```

2. Mock

▼ **Mockito**

Cria instancias fakes para evitar aclopamento na hora de executar os testes

▼ **Alguns metodos**

when("Quando chamar metodo").thenReturn("retorno");

when("Quando chamar metodo").thenThrow("new exception esperada");

verify(mock).metodo(parametro) veirifica se o metodo foi executado

verify(mock, times(N)).metodo(parametro) verifica quantas vezes o metodo foi executado