# Java Developer

## Java Core

# Sumário

**Objetivos deste Módulo**

Ao final deste módulo, objetiva-se que o aluno adquira:

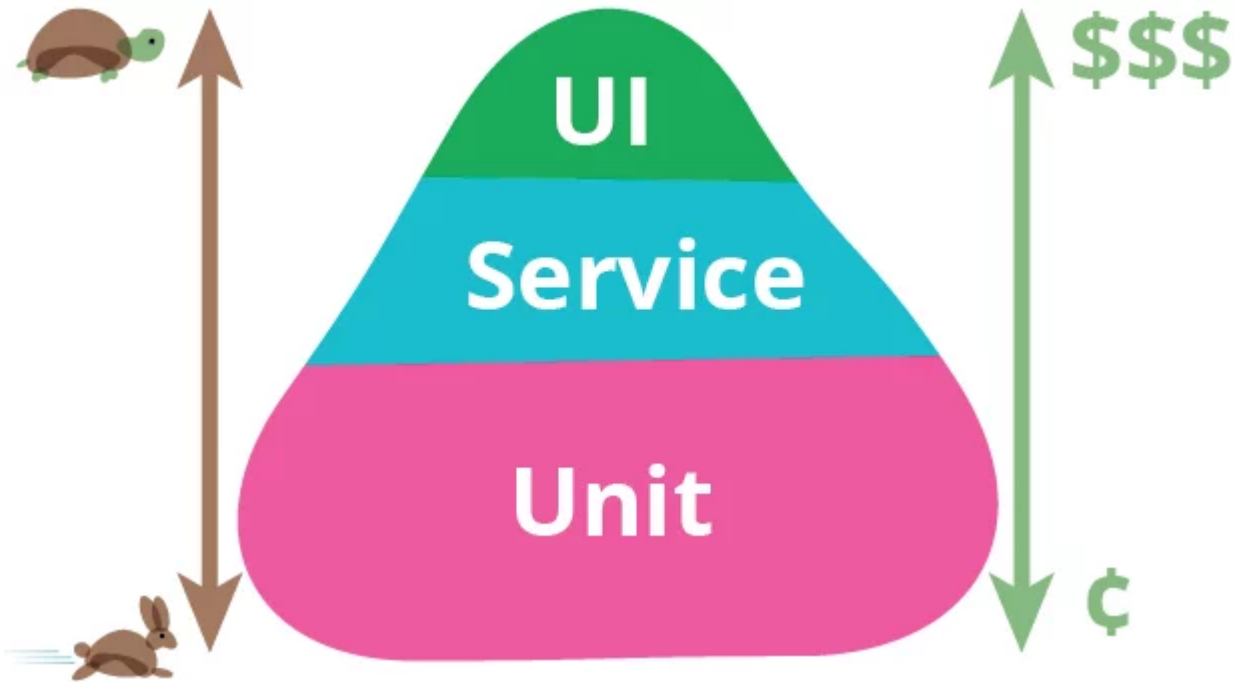
Conhecimentos:

- Pirâmide de testes
- Desenvolver testes unitários com JUnit
- Maven, Gradle
- Annotations

## 1. Pirâmide de testes

A "pirâmide de testes" é um conceito que representa uma estratégia de teste de software que sugere como os diferentes tipos de testes devem ser distribuídos em diferentes níveis de uma aplicação. Essa distribuição é frequentemente representada em forma de uma pirâmide, com os tipos de testes mais fundamentais na base da pirâmide e os mais abrangentes no topo.

A ideia é que a maioria dos testes deve se concentrar nos níveis mais baixos da pirâmide, pois esses testes são mais rápidos, mais fáceis de manter e mais baratos, enquanto os testes no topo são mais lentos, mais complexos, menos frequentes e mais caros.




**Testes unitários** são uma prática de teste de software que envolve a verificação de unidades individuais de código-fonte para garantir que funcionem corretamente. Uma unidade é a menor parte isolada de um programa que pode ser testada de forma independente. Geralmente, uma unidade corresponde a uma função ou método.

**Testes integrados**, também conhecidos como **testes de integração** (algumas literaturas chamam de camada de serviço), são uma categoria de testes de software que se concentra na verificação da interação e da integração entre diferentes partes de um sistema ou módulo. O objetivo principal dos testes integrados é garantir que os componentes individuais de um sistema funcionem corretamente quando trabalham juntos como um todo. Eles se situam em um nível intermediário da "pirâmide de testes".

Os **testes end-to-end (E2E) ou testes de interface do usuário (UI)** são um tipo de teste de software que avalia o comportamento de um aplicativo em um nível de sistema, simulando interações de usuário reais. Esses testes são projetados para verificar se o aplicativo funciona corretamente como um todo, desde a interface do usuário até o backend e todas as partes intermediárias. Os testes end-to-end são frequentemente executados para garantir que todo o fluxo de trabalho de uma aplicação seja testado em condições o mais próximas possível daquelas encontradas pelos usuários finais. Eles são usados para testar cenários completos de uso do aplicativo, que podem abranger várias páginas ou funcionalidades. Por

exemplo, um teste E2E pode verificar o processo de compra em um site de comércio eletrônico, desde a seleção de produtos até o pagamento.

## 2. Testes unitários

Aqui está um exemplo simples de teste unitário em Java usando **JUnit**. Vamos supor que temos uma classe chamada **Calculadora** com um método **somar** que queremos testar:

```
public class Calculadora {

    public int somar(int a, int b) {
        return a + b;
    }

}
```

Agora, podemos criar um teste unitário para esse método usando JUnit. Primeiro, certifique-se de que você tenha a biblioteca JUnit no seu projeto. Em seguida, crie uma classe de teste, por exemplo, **CalculadoraTest**:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculadoraTest {

    @Test
    public void testSomar() {

        // Arrange (Preparação)
        Calculadora calculadora = new Calculadora();

        // Act (Ação)
        int resultado = calculadora.somar(2, 3);

        // Assert (Asserção)
        assertEquals(5, resultado);
    }

}
```

Neste exemplo:

**@Test** é uma anotação (annotation) JUnit que indica que o método **testSomar** é um método de teste.

Na seção "**Arrange**", criamos uma instância da classe **Calculadora**.

Na seção "**Act**", chamamos o método **somar** da Calculadora e armazenamos o resultado em uma variável.

Na seção "**Assert**", usamos métodos como **assertEquals** para verificar se o resultado é igual ao valor esperado.

Agora, quando você executa este teste, ele verifica se a função somar da classe **Calculadora** está funcionando corretamente. Se o resultado for igual a 5 (como esperado neste caso), o teste passará; caso contrário, falhará, indicando que algo está errado na implementação da função.

Este é um exemplo muito simples de teste unitário em Java usando JUnit, mas ilustra os conceitos básicos. Em projetos mais complexos, você escreverá testes para cobrir uma variedade de cenários e casos de uso para garantir que seu código funcione corretamente em todas as situações.

Outros exemplos usando abordagem BDD (Behavior-Driven Development) para nomear seus testes…

When -> quando
Then -> então

```
@Test
public void whenUsuarioAutenticadoThenAcessoPermitido() {

    // Arrange (Preparação)
    AutenticacaoService authService = new AutenticacaoService();
    Usuario usuario = new Usuario("username", "senha");

    // Act (Ação)
    boolean acessoPermitido = authService.autenticarUsuario(usuario);

    // Assert (Asserção)
    assertTrue(acessoPermitido);
}

@Test
public void whenUsuarioNaoAutenticadoThenAcessoNegado() {

    // Arrange (Preparação)
    AutenticacaoService authService = new AutenticacaoService();
    Usuario usuario = new Usuario("username", "senhaIncorreta");

    // Act (Ação)
    boolean acessoPermitido = authService.autenticarUsuario(usuario);

    // Assert (Asserção)
    assertFalse(acessoPermitido);
}
```

Nesse exemplo, temos dois testes relacionados à autenticação de usuários:

**whenUsuarioAutenticadoThenAcessoPermitido**: Este teste verifica o cenário em que um usuário é autenticado com sucesso e o acesso é permitido. O "When" (Quando) corresponde à ação de autenticação, e o "Then" (Então) reflete a asserção de que o acesso deve ser permitido.

**whenUsuarioNaoAutenticadoThenAcessoNegado**: Este teste verifica o cenário em que a autenticação falha e o acesso é negado. O "When" (Quando) novamente descreve a ação de autenticação, enquanto o "Then" (Então) indica a asserção de que o acesso deve ser negado.

Ao seguir essa convenção de nomenclatura, os nomes dos testes comunicam claramente o comportamento esperado em termos das etapas "**Quando**" e "**Então**", tornando os testes mais compreensíveis e alinhados com o comportamento esperado do código. Certifique-se de adaptar os nomes dos testes de acordo com o contexto específico do seu sistema e do comportamento que você está testando.

E se estivermos trabalhando com números com casa decimal? Precisamos definir no assert um **delta**

```
// Arrange (Preparação)
Calculadora calculadora = new Calculadora();

// Act (Ação)
double resultado = calculadora.somar(2.9801, 3.2568123);

// Assert (Asserção)
assertEquals(6.2369, resultado, 0.01);
```

O propósito do parâmetro delta desse método é determinar o **valor máximo da diferença entre os números** expected (o valor esperado) e actual (o valor retornado no método somar) para que eles sejam considerados o mesmo valor.

Por exemplo, vamos supor que você tem uma rotina que retorna um número de ponto flutuante double. Você espera o valor 2.5, mas a rotina retorna 2.499999999999

Mais exemplos de testes - Teste unitário com JUnit

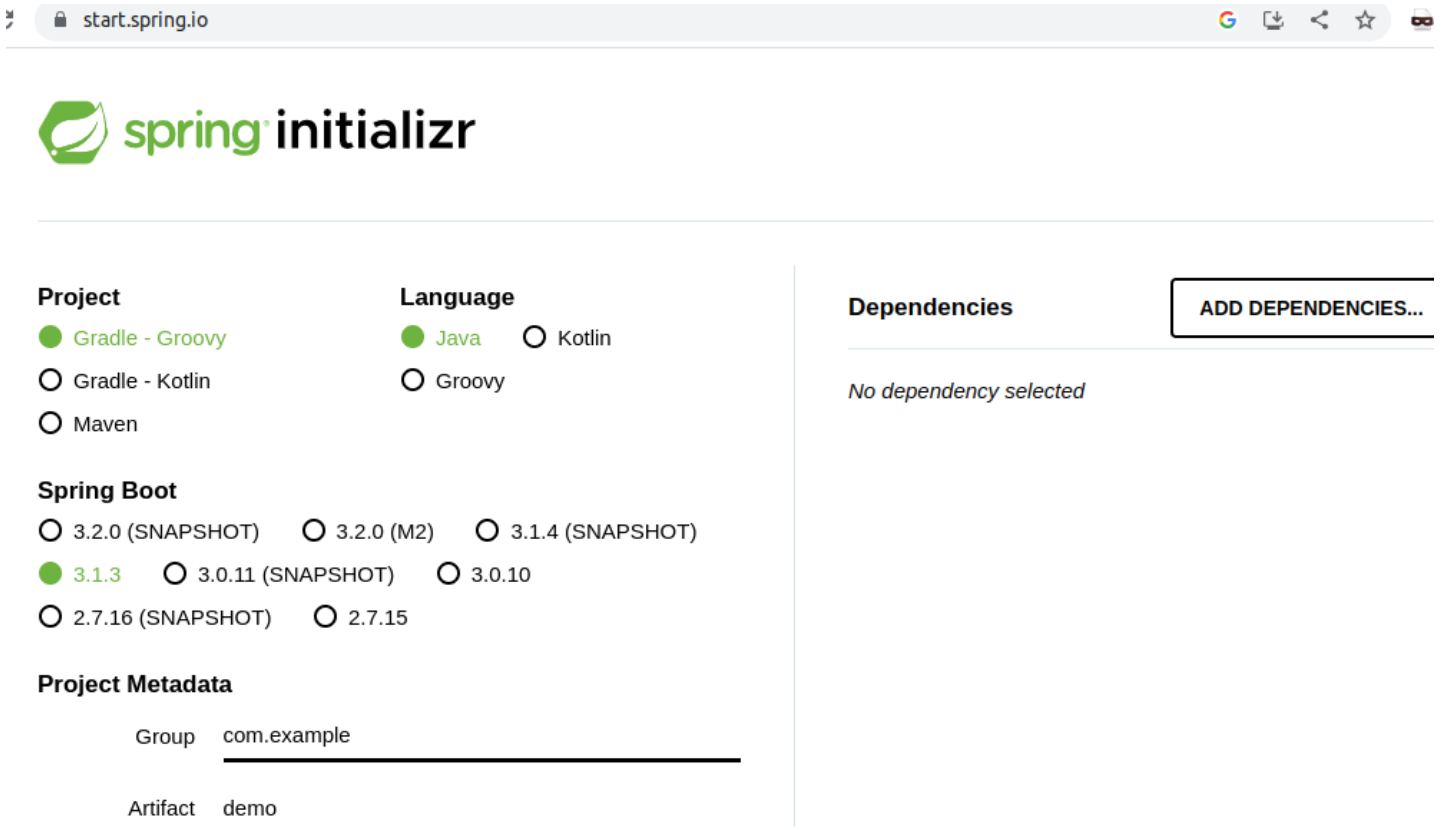
## 3. Maven x Gradle

O Maven e o Gradle são ferramentas de automação de construção de projetos de código aberto que são amplamente usados na indústria de desenvolvimento de software. São ferramentas de gerenciamento de construção que ajudam a automatizar tarefas de compilação, teste, empacotamento e distribuição de projetos de software.

| | Maven | Gradle |
|---|---|---|
| Linguagem de Configuração | XML - pom.xml | DSL (Domain-Specific Language) baseada em Groovy ou Kotlin - build.gradle |
| Flexibilidade | Embora seja altamente configurável, fazer tarefas personalizadas pode ser complicado. | É altamente flexível e permite uma personalização fácil. |
| Comunidade e Ecossistema | É mais antigo e tem uma comunidade grande e estável. | Está crescendo em popularidade e tem uma comunidade ativa. |

| | | |
|---|---|---|
| Tamanho e Performance | Os arquivos de configuração XML podem ser extensos e, em alguns casos, o Maven pode ser mais lento, especialmente em projetos grandes. | Os scripts de configuração do Gradle tendem a ser mais concisos, e o Gradle é conhecido por sua velocidade de construção em comparação com o Maven. |
| Exemplo de arquivo | pom.xml | build.gradle |

Para começar um projeto, podemos começar pela IDE ou pelo site spring initializr. Em ambos, podemos escolher o gerenciador de dependências, Maven ou Gradle.




## 4. Annotations

Em Java, as **anotações** (**annotations**) são metadados que podem ser adicionados ao código-fonte para fornecer informações adicionais sobre classes, métodos, campos, parâmetros e outros elementos do programa.

Imagine que você está escrevendo um livro, e em algumas páginas, você quer destacar partes importantes com um marcador de cor diferente. As "annotations" são como esses marcadores coloridos no seu código Java. Elas ajudam a destacar e explicar coisas importantes no código.

Então, em resumo, as "annotations" em Java são como notas especiais que você pode adicionar ao seu código para explicar o que está acontecendo e para que outras partes do seu programa ou ferramentas de

desenvolvimento entendam melhor o que você pretende fazer. Elas são úteis para tornar o código mais claro e expressivo.

Exemplos:

@Test
@RestController
@Service
@Repository

## 5. Criando primeiro projeto com testes unitários

Como buscar dependência: https://mvnrepository.com/search?q=junit

Criar projeto mencionado no item 2 com dependência Maven

## 6. Exercícios

1) No projeto que criamos uma calculadora, incluir testes unitários para cada operação (somar, subtrair, dividir, multiplicar)

2) No projeto que calculamos IMC, altere o método para retornar a classificação do imc (Magreza, Magreza, Sobrepeso, Obesidade, Obesidade grave) e crie um teste unitário para cada cenário.

3) Selecione algum outro projeto que você desenvolveu e crie teste unitário.