Java Básico

Anatomia das classes

A escrita de códigos de um programa é feito através da composição de palavras pré-definidas pela linguagem com as expressões que utilizamos para determinar o nome do nossos arquivos, classes, atributos e métodos.

É recomendado que nos projetos, toda a implementação seja escrita em inglês.

```
public class myClass{}
```

- 99% das classes se iniciarão com **public class**;
- Toda classe precisa de um nome;
- Todo o conteúdo fica dentro do copro {}.

```
public class myClass{
	public static void main (String[] args){

	}
}
```

- Dentro de uma aplicação, recomenda-se que somente uma classe possua o método main, responsável por rodar todo o programa;

Nomeclatura

- **Arquivo .java:** Todo arquivo .java deve começar com letra MAIÚSCULA. Se a palavra for composta, a segunda palavra deve também ser maiúscula, exemplo:

  `Calculadora.java`, `CalculadoraCientifica.java`

- **Nome da classe no arquivo**: A classe deve possuir o mesmo nome do arquivo.java, exemplo:

```
// arquivo CalculadoraCientifica.java

public class CalculadoraCientifica {

}
```

Nome da variável

- Deve ser escrita com letra minúscula, e caso tenha uma outra palavra, essa segunda palavra deverá ser inicializada com letra maiúscula.
- Deve obrigatoriamente se iniciar por uma letra (preferencialmente), _ ou $, jamais com número
- Deve iniciar com uma letra minúscula (boa prática – ver abaixo)
- Não pode conter espaços

- Não podemos usar palavras-chave da linguagem
- O nome deve ser único dentro de um escopo
- Deve conter apenas letras, _ (underline), $ ou os números de 0 a 9

Declarando variáveis e métodos

variável: Tipo nomeBemDefinido = Atribuição

método: TipoRetorno nomeObjetivoNoInifiniritivo Parametro(s)
ex: int somar (int numeroUm, int numeroDois){}

POO

classes: Toda a estrutura de código, na linguagem Java é distribuído em arquivos, com extensão **.java** denominados de **classe**.

Identificador, **Características** e **Comportamentos**.

- **Classe** (class): A estrutura e/ou representação que direciona a criação dos objetos de mesmo tipo.
- **Identificador** (identity): Propósito existencial aos objetos que serão criados.
- **Características** (states)**:** Também conhecido como **atributos** ou **propriedades**, é toda informação que representa o estado do objeto.
- **Comportamentos** (behavior)**:** Também conhecido como **ações** ou **métodos**, é toda parte comportamental que um objeto dispõe.
- **Instanciar** (new)**:** É o ato de criar um objeto a partir de estrutura, definida em uma classe.

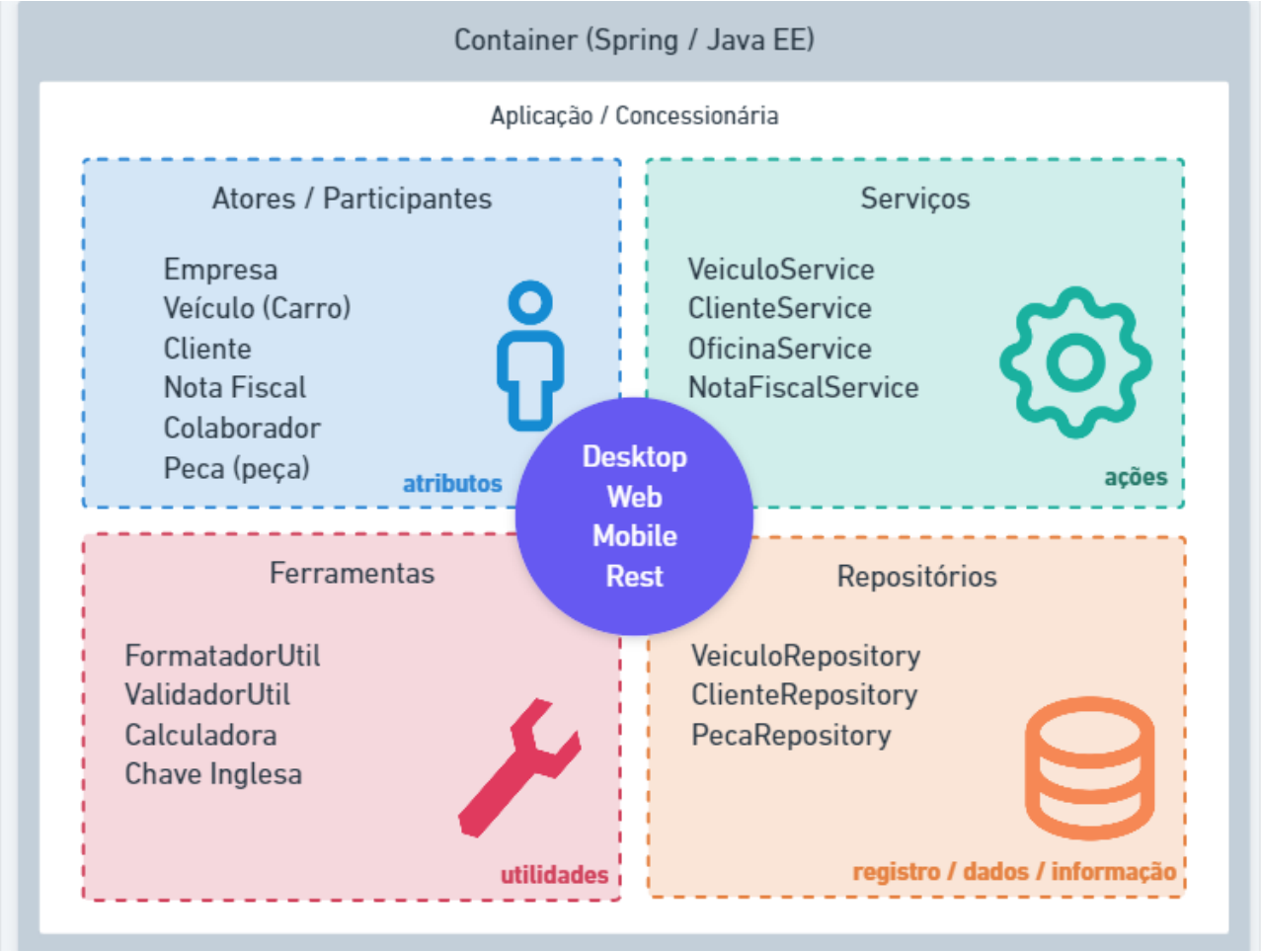
classe de modelo (model): representa estrutura de domínio da aplicação, como Cliente, Pedido, notafiscal.

Service: classes que contém regras de negócio da aplicação e validação do sistema.

Repository: classes que contém uma integração com bd.

Controller: classes que possuem a finalidade de disponibilizar alguma comunicação externa com à nossa aplicação, tipi http ou webservices.

Util: classe que contém recursos comuns à toda nossa aplicação.



Pacotes:

Vamos imaginar, que sua empresa se chama **Power Soft** e ela está desenvolvendo software comercial, governamental e um software livre ou de código aberto. Abaixo teríamos os pacotes sugeridos conforme tabela abaixo:

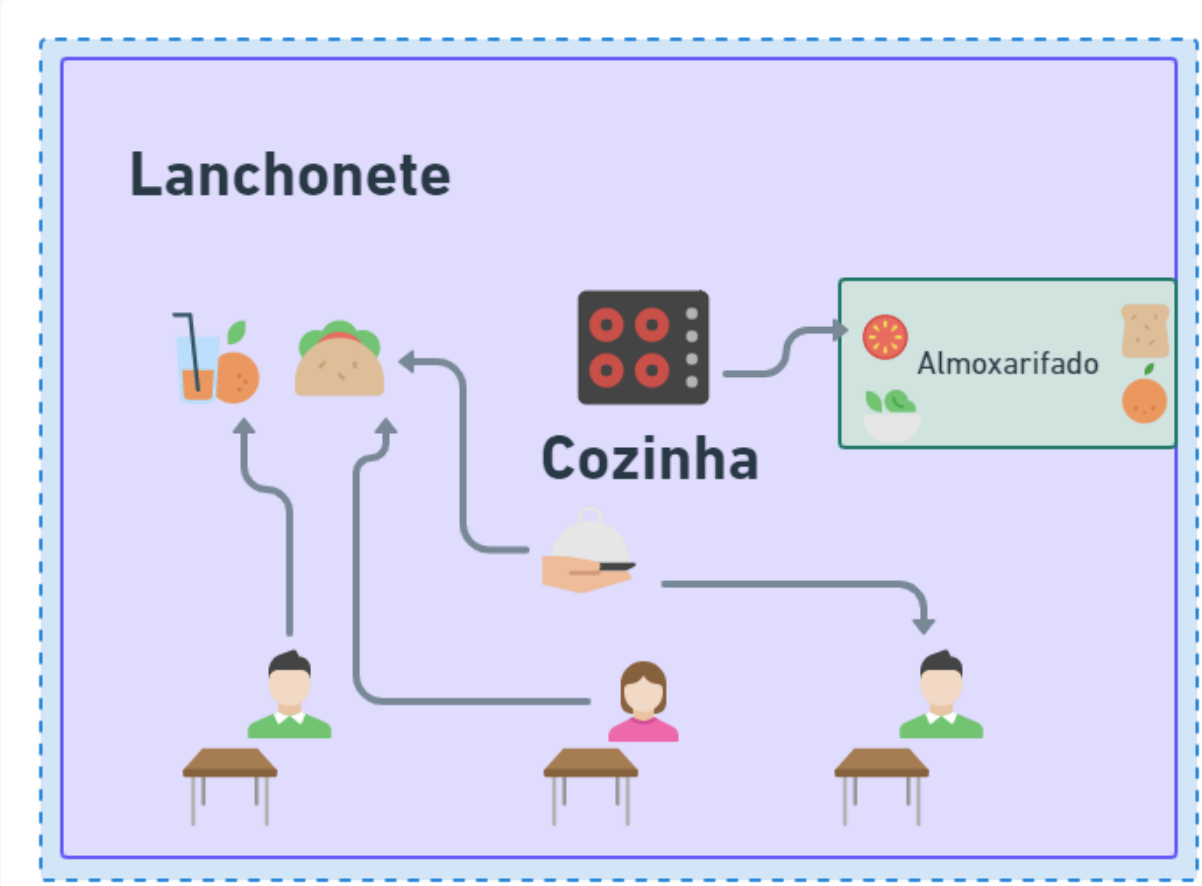
- **Comercial** : com.powersoft;
- **Governamental** : gov.powersoft;
- **Código aberto**: org.powersoft.

Visibilidade dos recursos:

Em Java, utilizamos três palavras reservadas e um conceito default (sem nehuma palavra reservada) para definir os quatro tipo de visibilidade de atributos, métodos e até mesmo classes, no que se refere ao acesso por outras classes. Iremos ilustrar do mais visível, ao mais restrito tipo de visibilidade nos arquivos em nosso projeto.

Modificador public

Como o próprio nome representa, quando nossa classe, método e atributo é definido como public, qualquer outra classe em qualquer outro pacote, poderá visualizar tais recursos.

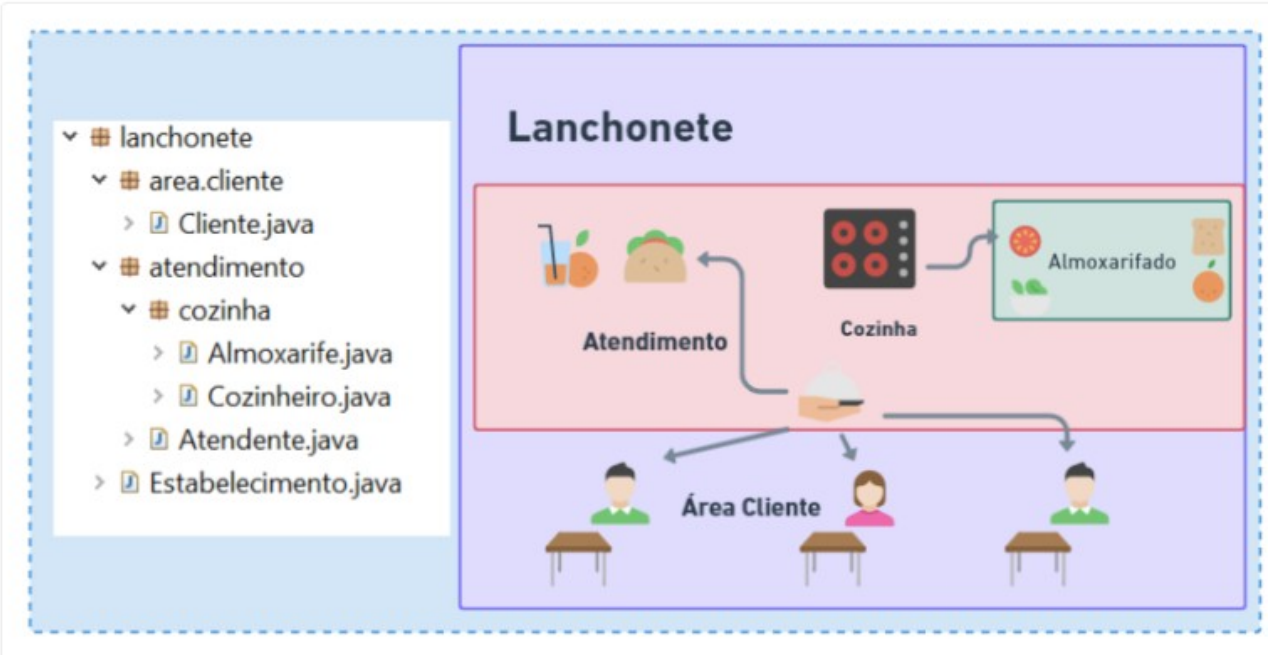


Modificador default

O modificador **`default`,** está fortemente associado a organização das classes por pacotes, algumas implementações, não precisam estar disponíveis por todo o projeto, e este modificador de acesso, restringe a visibilidade por pacotes.

Dentro do pacote **`lanchonete`**, iremos criar dois sub-pacotes para representar a divisão do estabelecimento.

- **lanchonete.atendimento.cozinha**: Pacote que contém classes, da parte da cozinha da lanchonete e atendimentos.
- **lanchonete.area.cliente**: Pacote que contém classes, relacionadas ao espaço do cliente.



Modificador private

Depois de reestruturar nosso estabelecimento (projeto), onde temos as divisões (pacotes), espaço do cliente e atendimento, chegou a hora de uma reflexão sobre visibilidade ou modificadores de acesso.

Conhecemos as ações disponíveis nas classes `Cozinheiro, Almoxarife, Atendente e Cliente`, mesmo com a organização da visibilidade por pacote, será que realmente estas classes precisam ser tão explícitas?

- Será que o `Cozinheiro` precisa saber que\como o `Almoxarife` controla as entradas e saídas ?
- Que o `Cliente` precisa saber como o `Atendente` recebe o pedido, para servir sua mesa ?
- Que o `Atendente` precisa saber que antes de pagar, o `Cliente` consulta o saldo no App ?

Getter e Setter

Seguindo a convenção Java Beans:

Os métodos "Getters" e "Setters" são utilizados para buscar valores de atributos ou definir novos valores de atributos, de instâncias de classes.

O método **Getter,** retorna o valor do atributo especificado.

O método **Setter,** define outro novo valor para o atributo especificado.

Construtores

Sabemos que, para criar um objeto na linguagem Java, utilizamos a seguinte estrutura de código:

```
Classe novoObjeto = new Classe();
```

Desta forma, será criado um novo objeto na memória, este recurso também é conhecido como instanciar um novo objeto.

Muitas vezes, já queremos que na criação (instanciação) de um objeto, a linguagem já solicite para quem for criar este novo objeto, defina algumas propriedades essenciais.

Pilares da orientação a objeto:

A programação orientada a objetos, é bem requisitada no contexto das aplicações mais atuais no mercado, devido a possibilidade de reutilização de código e a capacidade de representação do sistema, ser muito mais próximo do mundo real.

Para uma linguagem ser considerada orientada a objetos, esta deve seguir o que denominamos como **Os quatro pilares da orientação a objetos**:

- **Encapsulamento:** Nem tudo precisa estar visível, grande parte do nosso algoritmo pode ser distribuído em métodos, com finalidades específicas que complementam uma ação em nossa aplicação.

  Exemplo: Ligar um veículo, exige muitas etapas para a engenharia, mas o condutor só visualiza a ignição, dar a partida e a “magia” acontece.

- **Herança:** Características e comportamentos comuns, podem ser elevados e compartilhados através de uma hierarquia de objetos.

  Exemplo: Um Carro e uma Motocicleta possuem propriedades como placa, chassi, ano de fabricação e métodos como acelerar e frear. Logo, para não ser um processo de codificação redundante, podemos desfrutar da herança criando uma classe **Veículo** para que seja herdada por **Carro** e **Motocicleta**.

- **Abstração:** É a indisponibilidade, para determinar a lógica de um ou vários comportamentos, em um objeto.

  Exemplo: **Veículo** determina duas ações como acelerar e frear, logo, estes comportamentos deverão ser abstratos, pois existem mais de uma maneira de se realizar a mesma operação. ver Polimorfismo.

- **Polimorfismo:** São as inúmeras maneiras de se realizar uma mesma ação.

  Exemplo: Veículo determina duas ações como acelerar e frear, primeiramente, precisamos identificar se estaremos nos referindo a **Carro** ou **Motocicleta,** para determinar a lógica de aceleração e frenagem dos respectivos veículos.

Interface

A medida que vão surgindo novas necessidades, novos equipamentos (objetos), que nascem para atender as expectativas de oferecer ferramentas com finalidades bem específicas, como por exemplo: Impressoras, Digitalizadoras, Copiadoras e etc.

Observem que não há uma especificação de marca, modelo e ou capacidades de execução das classes citadas acima, isto é o que consideramos o nível mais abstrato da orientação a objetos, que denominamos como **interfaces**.

Java não permite múltiplos extends, sendo necessário usar interface para tal objetivo.

Collections:

Uma coleção (collection) é uma estrutura de dados que serve para agrupar muitos elementos em uma única unidade, estes elementos precisão ser Objetos.

•Uma Collection pode ter coleções homogêneas e heterogêneas, normalmente utilizamos coleções homogêneas de um tipo especifico.

•O núcleo principal das coleções é formado pelas interfaces da figura a abaixo, essas interfaces permitem manipular a coleção independente do nível de detalhe que elas representam.

•Temos quatro grandes tipos de coleções: `List`(lista), `Set`(conjunto), `Queue`(fila) e `Map`(mapa), a partir dessas interfaces, temos muitas subclasses concretas que implementam varias formas diferentes de se trabalhar com cada coleção.

Generics Type:

- Um tipo genérico é uma classe genérica ou uma interface que é parametrizada em relação a tipos.

```java
/**
Versão genérica da classe Box.
@param <T> o tipo do valor sendo armazenado
*/
public class Box<T> {
        // T representa "Type" (tipo)
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Uma variável de tipo pode ser qualquer tipo não primitivo que você especificar: qualquer tipo de classe, qualquer tipo de interface, qualquer tipo de array ou até mesmo outra variável de tipo.

Vantagens:

1. Segurança do tipo de dados: O uso de generics garante que apenas objetos de um tipo específico possam ser adicionados à coleção, evitando erros de tipo e garantindo que você esteja lidando com os dados corretos.
2. Código mais legível: Ao usar generics, você pode especificar o tipo de dados esperado ou retornado pela coleção, o que torna o código mais fácil de entender e ler.
3. Detecta erros mais cedo: O compilador verifica se você está usando os tipos corretos durante a compilação, ajudando a identificar erros de tipo antes mesmo de executar o programa.
4. Reutilização de código: Com generics, você pode criar classes e métodos genéricos que funcionam com diferentes tipos de coleções, evitando a necessidade de duplicar código para cada tipo específico.
5. Melhor desempenho: O uso de generics pode melhorar o desempenho, pois evita a necessidade de conversões de tipo desnecessárias e permite que o compilador otimize o código com base no tipo especificado.

Comparable x Comparator:

Comparable:

- Fornece uma única sequência de ordenação. Em outras palavras, podemos ordenar a coleção com base em um único elemento, como id, nome e preço.
- afeta a classe original, ou seja, a classe atual é modificada.
- fornece o método `compareTo()` para ordenar elementos.
- está presente no pacote `java.lang`.
- Podemos ordenar os elementos da lista do tipo `Comparable` usando o método `Collections.sort(List)`.

Comparator:

- Fornece o método compare() para ordenar elementos;
- fornece múltiplas sequências de ordenação. Em outras palavras, podemos ordenar a coleção com base em múltiplos elementos, como id, nome, preço, etc;
- não afeta a classe original, ou seja, a classe atual não é modificada;
- está presente no pacote java.util.
- Podemos ordenar os elementos da lista do tipo Comparator usando o método Collections.sort(List, Comparator).

List Interface:

- A interface list é uma coleção ordenada que permite a inclusão de elementos duplicados.
- É um dos tipos de coleção mais utilizados em Java. As classes mais comuns são ArrayList e Linkedlist.

- A list se assemelha a uma matriz com comprimento dinâmico.
- Fornece métodos úteis para adicionar elementos em posições específicas, remover ou substituir elementos com base no índice e obter sublistas usando índices.
- A classe Collections fornece algoritmos úteis para manipulação de List, como ordenação (sort), embaralhamento (shuffle), reversão (reverse) e busca binária (binarySearch).

Set interface:

- A interface Set é uma coleção que não pode conter elementos duplicados.
- Essa interface representa o conceito matemático de um conjunto e é usada para representar conjuntos, como um baralho de cartas.
- A plataforma Java possui três implementações de Set de uso geral: HashSet, TreeSet e LinkedHashSet.
- A interface Set não permite acesso aleatório a um elemento na coleção.
- Para percorrer os elementos de um Set, você pode usar um iterador ou um loop foreach.

Map interface:

- A interface Map é usada para mapear dados na forma de chaves e valores.
- O Map do Java é um objeto que mapeia chaves para valores.
- Um Map não pode conter chaves duplicadas: cada chave pode mapear no máximo um valor.
- A plataforma Java possui três implementações gerais de Map: HashMap, TreeMap e LinkedHashMap.
- As operações básicas do Map são: put (inserir ou atualizar), get (obter), containsKey (verificar se contém uma chave), containsValue (verificar se contém um valor), size (obter o tamanho) e isEmpty (verificar se está vazio).

Ganhando produtividade com Stream API

A Streams API traz uma nova opção para a manipulação de coleções em Java seguindo os princípios da programação funcional.

Stream, trata-se de uma poderosa solução para processar coleções de maneira declarativa, ao invés da tradicional e burocrática forma imperativa.

Na forma imperativa, para realizar uma soma simples, por exemplo, o desenvolvedor tem que se preocupar não apenas com o que deve ser feito em cada elemento, isto é, com as regras associadas ao processamento dos elementos da lista, mas também com a maneira de realizar essa iteração.

```java
public class CarrinhoDeCompras {
  //atributos
  private List<Item> itemList;
  //construtor
  public CarrinhoDeCompras() {
    this.itemList = new ArrayList<>();
  }

  //método para calcular valor total dos itens sem utilizar o Stream API
  public double calcularValorTotal() {
    double valorTotal = 0d;
    if (!itemList.isEmpty()) {
      for (Item item : itemList) {
        double valorItem = item.getPreco() * item.getQuant();
        valorTotal += valorItem;
      }
      return valorTotal;
    } else {
      throw new RuntimeException("A lista está vazia!");
    }
  }
}
```

Combinada com as Expressões Lambda e Method reference, eles proporcionam uma forma diferente de lidar com conjuntos de elementos, oferecendo ao desenvolvedor uma maneira simples e concisa de escrever código que resulta em facilidade de manutenção e paralelização sem efeitos indesejados em tempo de execução.

```java
public class CarrinhoDeCompras {
  //atributos
  private List<Item> itemList;
  //construtor
  public CarrinhoDeCompras() {
    this.itemList = new ArrayList<>();
  }

  //método para calcular valor total dos itens utilizando o Stream API
  public double calcularValorTotal() {
    if (itemList.isEmpty()) {
      throw new RuntimeException("A lista está vazia!");
    }
    return itemList.stream()
        .mapToDouble(item -> item.getPreco() * item.getQuant())
        .sum();
  }
}
```

As operações na Stream API podem ser classificadas em duas categorias principais:

1.**Operações Intermediárias:** são aquelas que retornam uma nova Stream e permitem encadear várias operações, formando um pipeline de processamento de dados. São elas:

•filter(Predicate<T> predicate): Filtra os elementos da Stream com base em um predicado. Retorna uma nova Stream contendo apenas os elementos que atendem ao critério do predicado. Exemplo: stream.filter(n -> n > 5)

•map(Function<T, R> mapper): Transforma cada elemento da Stream usando a função especificada e retorna uma nova Stream contendo os elementos resultantes.
Exemplo: stream.map(s -> s.toUpperCase())

•sorted(): Classifica os elementos da Stream em ordem natural (se os elementos forem comparáveis) ou de acordo com um comparador fornecido. Exemplo: stream.sorted()

•distinct(): Remove elementos duplicados da Stream, considerando a implementação do método equals() para comparação. Exemplo: stream.distinct()

•limit(long maxSize): Limita o número de elementos da Stream aos maxSize primeiros elementos Exemplo: stream.limit(10)

•skip(long n): Pula os primeiros n elementos da Stream e retorna uma nova Stream sem eles. Exemplo: stream.skip(5)

2.**Operações Terminais:** são aquelas que encerram o pipeline e produzem um resultado final. São elas:

•forEach(Consumer<T> action): Executa uma ação para cada elemento da Stream.
Exemplo: stream.forEach(System.out::println)

•toArray(): Converte a Stream em um array contendo os elementos da Stream.
Exemplo: stream.toArray()

•collect(Collector<T, A, R> collector): Coleta os elementos da Stream em uma estrutura de dados específica, como uma lista ou um mapa. Exemplo: stream.collect(Collectors.toList())

•count(): Retorna o número de elementos na Stream. Exemplo: stream.count()

•anyMatch(Predicate<T> predicate): Verifica se algum elemento da Stream atende ao predicado especificado. Exemplo: stream.anyMatch(s -> s.startsWith("A"))

•allMatch(Predicate<T> predicate): Verifica se todos os elementos da Stream atendem ao predicado especificado. Exemplo: stream.allMatch(n -> n > 0)

•noneMatch(Predicate<T> predicate): Verifica se nenhum elemento da Stream atende ao predicado especificado. Exemplo: stream.noneMatch(s -> s.isEmpty())

•min(Comparator<T> comparator) e max(Comparator<T> comparator): Encontra o elemento mínimo e máximo da Stream, respectivamente, de acordo com o comparador fornecido.
Exemplo: stream.min(Comparator.naturalOrder()) ou stream.max(Comparator.naturalOrder( ))

•reduce(T identity, BinaryOperator<T> accumulator): Combina os elementos da Stream usando o acumulador especificado e retorna o resultado final. Exemplo: stream.reduce(0, (a, b) -> a + b)

Lambda

As expressões lambda permitem representar interfaces funcionais (interfaces com um único método abstrato) de forma mais concisa e possibilitam escrever código no estilo funcional.

As interfaces funcionais desempenham um papel crucial na programação funcional em Java, pois servem de base para o uso de expressões lambda.

Uma função lambda é uma função sem declaração, isto é, não é necessário colocar um nome, um tipo de retorno e o modificador de acesso. A ideia é que o método seja declarado no mesmo lugar em que será usado.

As funções lambda em Java tem a sintaxe definida como (argumento) -> (corpo).

```java
public class OrdenacaoPessoa {
  //atributo
  private List<Pessoa> pessoaList;

  //construtor
  public OrdenacaoPessoa() {
    this.pessoaList = new ArrayList<>();
  }

  public List<Pessoa> ordenarPorAltura() {
    if (!pessoaList.isEmpty()) {
      List<Pessoa> pessoasPorAltura = new ArrayList<>(pessoaList);
      pessoasPorAltura.sort((p1, p2) -> Double.compare(p1.getAltura(), p2.getAltura()));
      return pessoasPorAltura;
    } else {
      throw new RuntimeException("A lista está vazia!");
    }
  }
}
```

Method Reference

Method Reference é um novo recurso do Java 8 que permite fazer referência a um método ou construtor de uma classe (de forma funcional) e assim indicar que ele deve ser utilizado num ponto específico do código, deixando-o mais simples e legível.

Para utilizá-lo, basta informar uma classe ou referência seguida do símbolo “::” e o nome do método sem os parênteses no final.

```java
public class OrdenacaoPessoa {
  //atributo
  private List<Pessoa> pessoaList;

  //construtor
  public OrdenacaoPessoa() {
    this.pessoaList = new ArrayList<>();
  }

  public List<Pessoa> ordenarPorAltura() {
    if (!pessoaList.isEmpty()) {
      List<Pessoa> pessoasPorAltura = new ArrayList<>(pessoaList);
      pessoasPorAltura.sort(Comparator.comparingDouble(Pessoa::getAltura));
      return pessoasPorAltura;
    } else {
      throw new RuntimeException("A lista está vazia!");
    }
  }
}
```

Gerenciamento de dependências e build em Java com Maven

O que é Apache Maven

- Ferramenta para gerenciar build e dependências de um projeto.

Maven

- Endereça como o software foi construído e suas dependências através do POM(Project Object Model)
- Facilita a compreensão do desenvolvedor
- Fornecer informações de qualidade

COMANDOS:

- mvn archetype:generate -DgroupId=one.digitalinnovation -DartifactId=quick-start-maven -Darchetype=maven-archetype-quickstart -DinteractiveMode=false

Criação de um projeto maven.

- Mvn compile;
- mvn test;
- mvn package;
  cria arquivo .jar
- mvn clean;
- mvn classpath

https://docs.google.com/presentation/d/1wudqWaBDK40QnBAYjuh4Q65dcC2wqLW_/edit#slide=id.p117

Testes de sofware

Testes de software são processos utilizados para verificar e validar o funcionamento de um programa. Eles envolvem a execução de um software com a intenção de encontrar erros, verificar se o sistema atende aos requisitos especificados, garantir que funcione corretamente sob diferentes condições e validar a qualidade geral do produto. Os testes de software podem ser classificados em várias categorias, como testes funcionais, testes de desempenho, testes de segurança, entre outros, e podem ser realizados manualmente ou de forma automatizada.

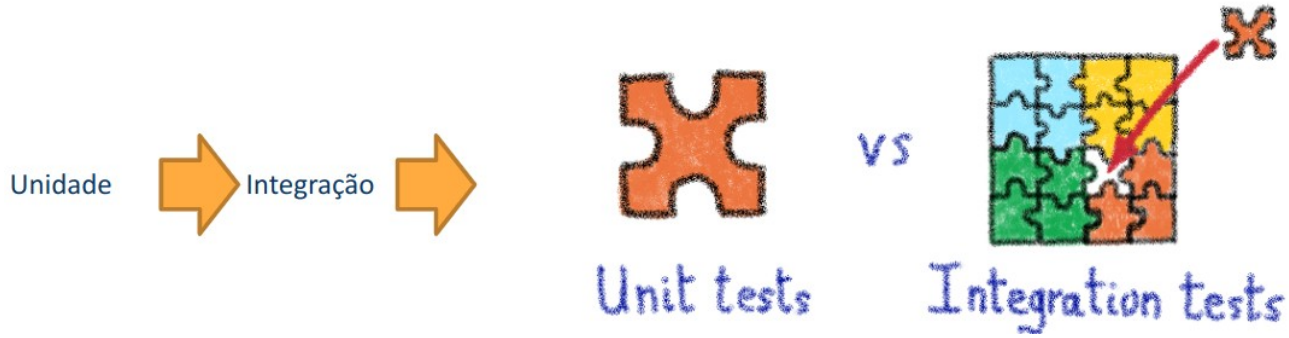
Níveis de teste

Testes de unidade

Testes de unidade são um tipo de teste de software focado em verificar o funcionamento correto de partes individuais e isoladas de um código, conhecidas como unidades. Geralmente, uma unidade é a menor parte testável de um aplicativo, como uma função, método ou classe. O objetivo dos testes de unidade é assegurar que cada unidade do software opere conforme esperado de forma independente. Eles são tipicamente escritos e executados por desenvolvedores durante a fase de desenvolvimento, facilitando a identificação e correção de erros de forma precoce e contribuindo para a manutenção de um código de alta qualidade.

Teste de integração

Teste de integração é um tipo de teste de software que visa verificar a interação e a integração entre diferentes unidades ou módulos de um sistema. Enquanto os testes de unidade se concentram em partes isoladas do código, os testes de integração garantem que essas partes funcionem corretamente quando combinadas. O objetivo é identificar problemas que possam surgir na interação entre os componentes, como interfaces incorretas, problemas de comunicação ou integração de dados. Esses testes ajudam a assegurar que os módulos individuais, quando integrados, operem de acordo com os requisitos especificados e que o sistema funcione de forma coesa.



Testes de sistema

Testes de sistema são um tipo de teste de software que envolve a verificação completa e a validação de um sistema integrado para garantir que ele atenda aos requisitos especificados. Este tipo de teste avalia o sistema como um todo, incluindo hardware, software, redes e outros componentes integrados, em um ambiente que reflete as condições reais de uso.

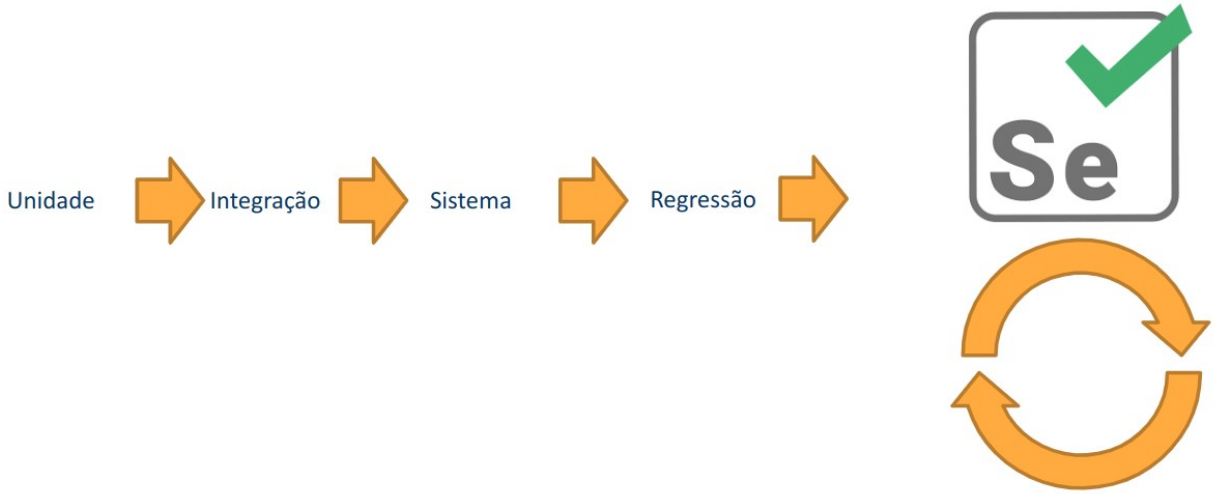
O objetivo dos testes de sistema é assegurar que o software funcione corretamente e de forma confiável em todas as suas funcionalidades, incluindo desempenho, segurança, usabilidade e

compatibilidade. Eles são geralmente realizados após os testes de unidade e de integração, e antes dos testes de aceitação pelo usuário.

Testes de regressão

Testes de regressão são um tipo de teste de software que têm como objetivo garantir que alterações no código, como novas funcionalidades, correções de bugs ou melhorias, não introduzam novos erros em partes do software que já funcionavam corretamente. Eles envolvem a reexecução de conjuntos de testes previamente realizados para verificar se o comportamento do software permanece consistente após as mudanças.

O processo de testes de regressão pode ser automatizado, o que é comum em ciclos de desenvolvimento contínuo, onde as alterações são frequentes. A automação ajuda a executar rapidamente um grande número de testes e detectar regressões de forma eficiente.



Níveis de teste

Técnicas de teste

Caixa branca**,** Também conhecido como Teste Estrutural:

- Validar dados, controles, fluxos, chamadas
- Garantir a qualidade da implementação
- Níveis: Unidade, Integração, Regressão

Caixa preta:

- Teste funcional
- Verificar saídas usando vários tipos de entrada
- Teste sem conhecer a estrutura interna do software
- Níveis: Integração, Sistema, Aceitação

Caixa cinza:

- Mescla técnicas de Caixa branca e Caixa Preta
- Analisa parte lógica e também funcionalidade
- Exemplo: Ter acesso a documentação do funcionamento do código
- Engenharia Reversa

Testes não funcionais

Testes não funcionais estão ligados a requisitos não funcionais:

- Comportamento do Sistema
- Performance
- Escalabilidade
- Segurança
- Infraestrutura

Exemplo: “Qual Plataforma o Sistema deverá rodar ?”

Testes de carga

"O teste de carga é realizado para verificar qual o volume de transações, acessos simultâneos ou usuários que um servidor/software/sistema suporta." Alguns pontos de atenção:

- Objetivos para clareza de resultados
- Ambiente
- Cenários
- Execução de testes
- Análise de resultado

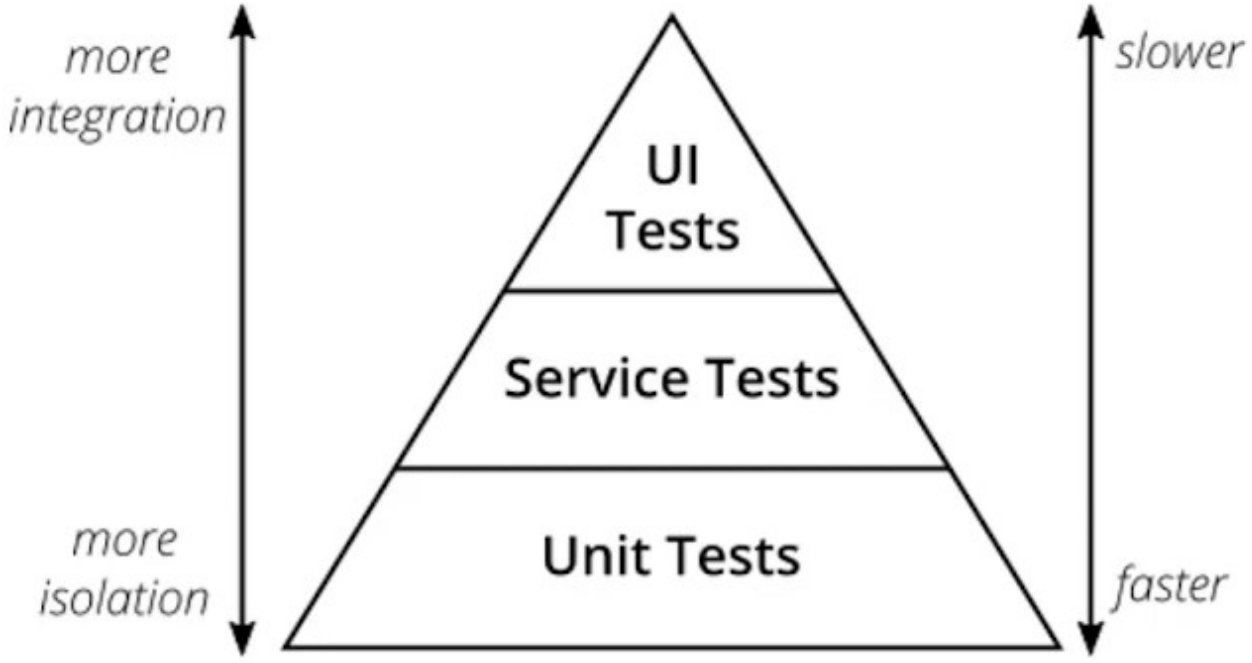
Teste de stress

Teste de stress é realizado para submeter o software a situações extremas. Basicamente, o teste de stress baseia-se em testar os limites do software e avaliar seu comportamento. Assim, avalia-se até quando o software pode ser exigido e quais as falhas (se existirem) decorrentes do teste.

Testes de segurança

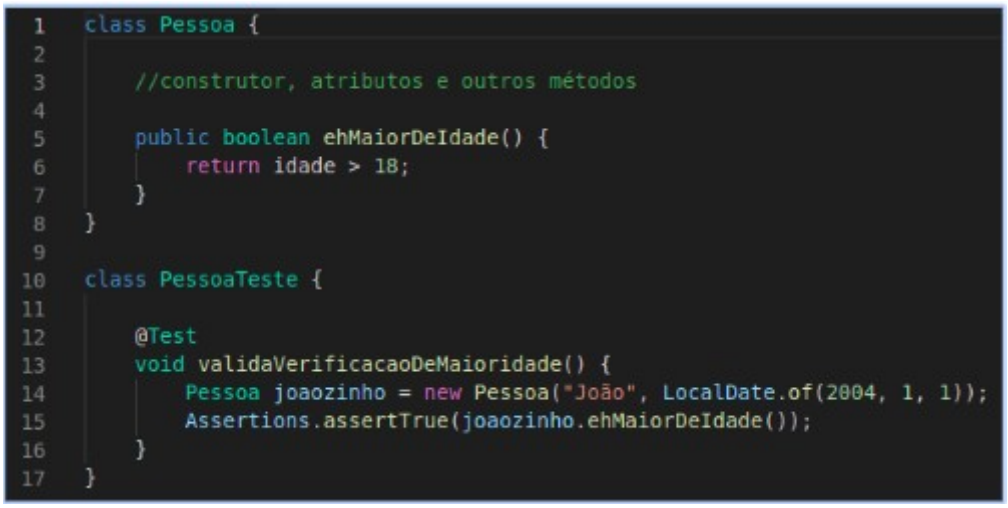
O teste de segurança é um processo crítico de segurança cibernética que visa detectar vulnerabilidades em sistemas, software, redes e aplicativo

Pirâmide de testes



Testes unitários

- Também chamado de testes de unidade;
- Testar a menor unidade de código possível;
- Unidade: função, método, classes;
- Testar uma aplicação na sua menor parte;
- Geralmente escrito em tempo de desenvolvimento.



```java
class Pessoa {

    //construtor, atributos e outros métodos

    public boolean ehMaiorDeIdade() {
        return idade > 18;
    }
}

class PessoaTeste {

    @Test
    void validaVerificacaoDeMaioridade() {
        Pessoa joaozinho = new Pessoa("João", LocalDate.of(2004, 1, 1));
        Assertions.assertTrue(joaozinho.ehMaiorDeIdade());
    }
}
```

Mockito

MockTests

Mock testes são uma técnica de teste de software onde objetos simulados (mocks) são usados para imitar o comportamento de objetos reais de forma controlada. Eles são usados para isolar a unidade de código sendo testada, permitindo que o teste se concentre apenas na funcionalidade da unidade, sem interferências de dependências externas. Mocks podem ser programados para retornar valores específicos ou para verificar se métodos foram chamados com parâmetros esperados.