***RESUMO ESTUDO PROGRAMAÇÃO EM JAVA***

**ÍNDICE**

**01) - Construtor** - *(livro JAVA COMO PROGRAMAR Ed.10)*

**02) - POO - Programação Orientada a Objetos**

**03) - Variáveis e Constantes**

**03.1) - Arrays**

**03.2) - Collections**

*1 - Set*

*2 - List*

*3 - Map*

*4 - Queue*

*5 - Stack*

**04) - Métodos**

**05) - Encapsulamento**

**06) - Wrapper Class Integer**

**07) - Loop for Aprimorado**

**08) - ArrayList**

**09) - API JAVA**

**10) - Herança e Níveis de Acesso**

*1 - Níveis de Acesso*

*2 - UpCasting e DownCastig*

*3 - Sobreposição ou sobrescrita*

*4 - Palavra reservada Super*

*5 - Palavra chave Final em Classes e Métodos*

**11) - String.format**

**12) - Formatação de Datas**

**13) - GUI - Graphic User Interface**

**14) - Método toString**

**15) - Classe Enumeração e Composição**

**16) - Classe StringBuilder**

**17) - Polimorfismo**

**18) - Classes Abstratas**

**19) - Métodos Abstratos**

**20) - Métodos Vários**

**21) - Tratamento de Exceções**

*1 - Estrutura try - catch*

*2 - Método printStackTrace() - Pilha de chamadas de métodos*

*3 - Bloco finally*

**22) - Classes Várias**

*1 - Classe File*

*2 - Classe Scanner*

*3 - Classe IOException*

*4 - Classe FileReader*

*5 - Classe BufferedReader*

*6 - Bloco try-with-resources*

*7 - FileWriter e BufferedWriter*

*8 - Manipulando pastas com File*

*9 - Informações do caminho do arquivo*

*10 - Classe SimpleDateFormat*

**23) - Interface**

*1 - Inversão de controle*

*2 - Injeção de dependência*

*3 - Herança vs Interface*

*4 - Herança múltipla e o problema do diamante*

*5 - Interface Comparable*

*6 - Default methods ou métodos padrão*

**24) - Generics, Set e Maps**

*1 - Generics*

**25 - Programação Funcional e Expressões Lambda**

*1 - Cálculo Lambda*

*2 - Expressão lambda*

*3 - Interface funcional Predicate*

*4 - Interface funcional Consumer*

*5 - Interface funcional Function*

*6 - Stream*

**26) - JDBC - Java Database Connectivity**

*1) - API padrão do Java para acesso a dados*

*2) - Criar uma conexão com banco de dados*

*3) - Recuperar dados*

*4) - Inserir dados*

*5) - Atualizar dados*

*6) - Deletar dados*

*7) - Transação segura*

*8) - Padrão de projeto DAO - Data Access Object*

**100) - Bônus Git e Github**

*1) - Configurar chave SSH para o Github*

*2) - Comandos no terminal Git Bash*

*3) - Passo a passo: salvar primeira versão de um projeto no Github*

*4) - Passo a passo: salvar uma nova versão do projeto no Github*

*5) - Passo a passo: clonar/modificar projeto de um repositório remoto que você tem permissão para alterar*

*6) - Arquivo .gitignore*

*7) - O que fazer quando abre o editor VIM*

*8) - Como resolver push rejeito por históricos diferentes*

*9) - Resolvendo pull com conflito*

*10) - Como sobrescrever um histórico no Github*

*11) - Como apontar o projeto para outro repositório remoto no Github*

**101) - Tópicos a pesquisar** *(citados em livros mas não apresentados)*

**1) - CONSTRUTOR** *(livro JAVA COMO PROGRAMAR Ed.10)*

**1 - Construtor**

- É semelhante a um método, mas que é chamado automaticamente pelo operador **new** para inicializar as variáveis de instância de um **objeto** da Classe quando este é criado.

- Os construtores ***não são herdados***, então uma subclasse não herda o construtor da superclasse Object. Mas os construtores de uma superclasse continuam disponíveis para serem chamados pelas subclasses.

- De fato, o Java requer que a ***primeira tarefa de qualquer construtor*** de subclasse seja chamar o construtor de sua superclasse direta, explícita ou implicitamente *(se nenhuma chamada de construtor for especificada),* para ***assegurar*** que as ***variáveis de instância herdadas*** da superclasse ***inicializem adequadamente***.

- Na ***sintaxe*** de chamada de construtor de superclasse usa-se a palavra-chave ***super*** seguida por um par de parênteses contendo os argumentos que serão usados para inicializar as variáveis de instância, ex.:

***- super*** *(primeiroNome, segundoNome, totalVendas, taxaComissão);*

**2 - Construtores e Métodos**

- Uma diferença importante entre **construtores** e **métodos** é que os construtores ***não podem retornar valores,*** portanto, não podem especificar um tipo de retorno (nem mesmo void). Normalmente, os construtores são declarados public — mais adiante no livro explicaremos quando usar **construtores** **private**.

**3 - Chamada de Construtor** - O Java **requer** uma chamada de **construtor** para cada objeto que é criado, então esse é o ponto ideal para inicializar variáveis de instância de um objeto.

**4 - Nomear um Construtor**

- Um **construtor** ***deve ter o mesmo nome da classe*** em que está especificado.

**5 - O que Evitar**

- Embora seja possível, ***não chame métodos a partir de construtores***. Este aspecto está explicado no *Capítulo 10, Programação orientada a objetos do livro Java Como Programar Ed.10:* *polimorfismo e interfaces.*

**6 - Declaração de Construtores**

- Os **Construtores** podem ser **Implícitos** ou **Explícitos**:

- **Implícitos** - significa que se você não declarar um Construtor ele será criado pela compilador*.* ***Por exemplo na criação do objeto:*** **ContaBanco minhaConta = new ContaBanco ();** foi usado a palavra chave ***new*** para criar um objeto **ContaBanco**. Os parênteses vazios depois de “**new** **ContaBanco**” indicam uma chamada para o ***construtor padrão da classe*** — em qualquer classe que não declare explicitamente um construtor, o compilador fornece um tipo padrão (que sempre ***não tem parâmetros***). Quando uma classe tem somente o construtor padrão, as variáveis de instância da classe são inicializadas de acordo com seus valores padrões.

- **Explícito** - significa que você pode criar seu próprio Construtor dentro da Classe para inicializar as variáveis de instância *(atributos)* da Classe. Se você declarar um construtor para uma classe, o ***compilador*** não criará um construtor padrão para ela.

**7 - Construtores na UML** - O diagrama de classe UML da Figura 3.7 modela a classe Account da Figura 3.5, que tem um construtor com um parâmetro **name** de **String**. Assim como as operações, a UML modela construtores no *terceiro* compartimento de um diagrama de classe. Para distinguir entre um construtor e as operações de uma classe, a UML requer que a palavra “constructor” seja colocada entre aspas francesas «construtor» antes do nome do construtor. É habitual listar construtores *antes* de outras operações no terceiro compartimento.

**2) - POO - Programação Orientada a Objetos**

**1 - Estrutura Java-Poo** - Um programa Java nada mais é do que uma pilha de classes interrelacionadas.

**2 - Conceitos chave de POO**

- **Classes** - Quando você vai projetar uma classe, pense nos objetos que serão criados com esse tipo de classe e considere: - as coisas que o objeto **conhece** são os **Atributos** ou **Variáveis de instância** *(estado do objeto).*

- as coisas que o objeto **faz** são os **Métodos** *(comportamento do objeto).*

- Objetos

- Métodos

- Sobreposição de métodos

- Herança

- Polimorfismo

**3 - Código de Teste** - O conceito de escrever o código de teste primeiro é uma das práticas da **XP - Extreme Programming** e ela pode tornar mais fácil e rápido a criação do seu código.

**4 - Classe Abstrata** - É aquela que possui métodos abstratos e não pode virar um objeto. Serve somente para efeito de herança.

- Uma classe Abstrata significa que ela deve ser estendida

**3) - Variáveis e Constantes**

**1 - Áreas de Memória** - O Java tem duas áreas de memória que é preciso conhecer: a PILHA e o HEAP.

- PILHA - Todas as variáveis locais são variáveis declaradas dentro de um método ou declaradas como parâmetro de um método, e residem na PILHA, no ponteiro correspondente ao método onde foram declaradas, inclusive as variáveis de referência de objeto quando são declaradas como variável local dentro de um método.

- HEAP - As variáveis de instância são declaradas dentro de uma classe e fora de qualquer método. Todos os objetos residem no HEAP, independentemente de a referência ser uma variável local ou de instância.

**2 - Tipo e Nome** - Variáveis precisam ter um ***tipo*** e um ***nome***.

**3 - Versões de Variáveis** - Existem 02 versões e variáveis: ***primitivas*** e de ***referência***.

**4 - Tabela - Tipos de Variáveis:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Item** | **Família** | **Tipo Primitivo** | **Classe Invólucro (Wrapper)** | **Tamanho** | **Armazenamento** |
| 1 | Lógico | **boolean** | Boolean | 1 bit | true |
| 2 | Literal | **char** | Character | 8 bits | A' |
| 3 | Literal | **-** | String | 8 bits | "Java" (armazenam textos e números como texto e observe que na sintaxe o 'S' de String é maiúculo) |
| 4 | Inteiro | **byte** | Byte | 8 bits | -128 **até** +127 |
| 5 | Inteiro | **short** | Short | 16 bits | -32768 **até** +32767 |
| 6 | Inteiro | **int** | Integer | 32 bits | -2.147.483.648 até **<==>** +2.147.483.647 |
| 7 | Inteiro | **long** | Long | 64 bits | -9.223.372.036.854.775.808 **até** +9.223.372.036.854.775.807 |
| 8 | Real | **float** | Float | 32 bits | Armazena ns. com precisão decimal de até **07 casas** depois do ponto. (A sintaxe exige que se coloque a letra 'f' no final do número.) |
| 9 | Real | **double** | Double | 64 bits | Armazena ns. com precisão decimal de até **15 casas** depois do ponto. (A sintaxe NÃO exige letra no final do número.) |

**5 - Inicializar Variáveis de Instância**

- Caso você não inicialize uma variável de instância ela será automaticamente inicilizada para um valor padrão.

- **inteiros** 🡺 valor padrão = **0**

- **ponto flutuante** 🡺 valor padrão = **0.0**

- **booleanos** 🡺 valor padrão = **false**

- **referências** 🡺 valor padrão = **null**

**6 - Variáveis de Instância**

- Variáveis de instância são declaradas dentro de uma classe e nunca dentro de um método. Elas representam os ***“campos”*** que cada objeto individual tem (que podem ser preenchidos com valores diferentes para cada instância da classe). As variáveis de instância residem dentro do objeto a que pertencem.

**7 - Variáveis Locais**

- Variáveis locais são declaradas dentro de um método, inclusive como parâmetros do método.

- Elas são temporárias e só existem enquanto o método estiver na pilha, ou seja, enquanto o método não alcançar a chave de fechamento.

- Não recebem um valor padrão. Você pode declarar uma variável local sem inicializá-la, mas ao tentar usá-la receberá um erro do compilador. Portanto elas precisam ser inicializadas antes do uso.

**8 - Variáveis Estáticas *(static)***

- As variáveis estáticas são inicializadas quando uma classe é carregada, e há duas coisas garantidas na inicialização estática *(ver programa* ***public class Cap\_10\_01\_PlayerTestDrive*** *do livro Use a Cabeça Java):*

a)- As variáveis estáticas de uma classe são inicializadas antes de qualquer objeto da classe poder ser criado.

b) - As variáveis estáticas de uma classe são inicializadas antes de qualquer método estático da classe ser executado.

- As variáveis estáticas se não forem inicializadas recebem os valores padrão da mesma forma que as variáveis de instância, ou seja:

- **int** recebe: **‘0’ (zero)**

- **byte** recebe **‘0’ (zero)**

- **short** recebe: **‘0’ (zero)**

- **long** recebe: **‘0’ (zero)**

- **double** recebe: **’0.0’**

- **float** recebe: **‘0.0’**

- **boolean** recebe: **‘false’**

- **char** recebe: **‘null’**

- **string** recebe: **‘null’**

- O acesso às variáveis estáticas é feito da mesma forma como se acessa um método estático, ou seja, usando o nome da classe. Exemplo: **Player.playerCount** - onde **Player** é a classe e **playerCount** é a variável estática.

**9 - Constantes**

- O valor de uma constante não pode ser alterado e ela é identificada pela palavra reservada ***‘final’***.

- A palavra chave ***final*** não é exclusiva para ***constantes***, ela pode ser usada para modificar ***variáveis locais***, ***variáveis de instância*** e até mesmo ***parâmetros de métodos***. Em todos esses casos ela tem o mesmo significado: o valor não pode ser alterado. E você também pode usar ***final*** para ***impedir*** que alguém ***sobreponha um método*** ou ***crie uma subclasse***.

- Por convenção as constantes são nomeadas com letras MAIÚSCULAS.

- Ex.: ***double final TX\_COMISSÃO = 3.5.***

- Uma constante pode também ser definida como ‘estática’.

- Ex.: ***public static double final TX\_COMISSAO = 3.5.***

- Uma constante pode ser inicializada de dois modos:

- Atribuindo um valor quando ela é declada como mostrado no exemplo acima ou dentro de um bloco estático.

- Ex.: ***public class IndiceAumento{***

***public static final double TX\_COMISSAO;***

***static { TX\_COMISSAO = 3.5;***

***}***

***}***

**3.1) - Arrays**

- ***Array*** significa um arranjo de dados organizados em um *vetor* *(unidimensional)* ou em uma *matriz* de linhas e colunas *(multidimensional)* que pode armazenar vários vários dados, ao contrário da variável simples que pode armazenar somente um dado por variável.

- Para *diferenciar* a declaração de um array de uma *variável* *simples* é preciso inserir colchetes ‘[ ]’ após o tipo ou o nome que o identifica.

**1) - Caracteristicas de Array**

- **Estático** - Array é uma estrutura de dados *estática*, ou seja, possui tamanho fixo onde não se pode acrescentar novos elementos. O tamanho de um array é definido no momento em que ele é criado.

- **Homogêneo** - Possui um tipo de dados espécifico *(int, String, double, Integer, Double, etc).*

- **É um objeto** - Ou seja, possui *atributos* e *comportamentos* associados ao Array.

- Exemplo: *String[] nomes = new String[6];*

*nomes[3] = "Carlos";*

*nomes[0] = "William";*

*int[] num = new int[5];*

*num[0] = 10;*

*num[4] = 8;*

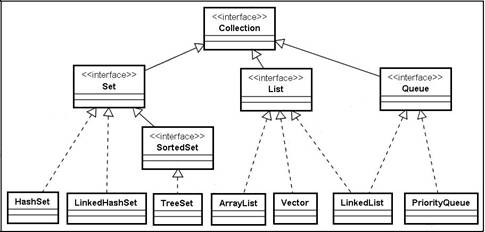
*String[] nomes2 = {“Maria”, “Bob”, “Alex”};*

*int[] num2 = {3, 4, 5, 6};*

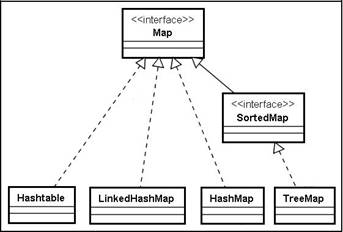
**3.2) - Collections** *(Coleções)*

*Collection* - está no topo da hierarquia. Não existe implementação direta dessa interface, mas ela define as operações básicas para as coleções, como adicionar, remover, esvaziar, etc.;

A **Figura 1** mostra a árvore da hierarquia de *interfaces* e *classes* da *Java* *Collections* *Framework* que são derivadas da *interface* *Collection*. O diagrama usa a notação da *UML*, onde as *linhas cheias* representam *extends* e as *linhas* *pontilhadas* representam *implements*.



A hierarquia da *Collections* Framework tem uma segunda árvore. São as *classes* e *interfaces* relacionadas a *mapas*, que não são derivadas de *Collection*, como mostra a **Figura 2**. Essas *interfaces*, mesmo não sendo consideradas *coleções*, podem ser manipuladas como tal.



**1) - Set -** *(Conjunto)* - *Set* - *interface* que define uma *coleção* que não permite elementos duplicados. A interface *SortedSet*, que *estende* *Set*, possibilita a classificação natural dos elementos, tal como a ordem alfabética.

**- Set <T>** - Representa um conjunto de elementos (*similar ao da Álgebra)*

- Não admite repetições

- Elementos não possuem posição

- Acesso, inserção e remoção de elementos são rápidos

- Oferece operações eficientes de conjunto: interseção, união, diferença.

**- Principais implementações:**

- ***HashSet*** - mais rápido (operações O(1) em tabela hash) e não ordenado

**- *TreeSet*** - mais lento (operações O(log(n)) em árvore rubro-negra) e ordenado pelo compareTo do objeto (ou Comparator)

***- LinkedHashSet*** - velocidade intermediária e elementos na ordem em que são adicionados

**- Alguns métodos importantes**:

***- add(obj), remove(obj), contains(obj)*** 🡺 Baseado em *equals* e *hashCode*. Se *equals* e *hashCode* não existir, é usada comparação de ponteiros

***- clear()***

***- size()***

***- removeIf(predicate)***

***- addAll(other)*** - união: adiciona no conjunto os elementos do outro conjunto, sem repetição

***- retainAll(other)*** - interseção: remove do conjunto os elementos não contitos em other

***- removeAll(other)*** - diferença: remove do conjunto os elementos contidos em other

**1.1) - HaschSet** - É uma classe

- Características - Não ordenado *(por padrão, mas existem formas de se ordenar).*

- Não indexado - não tem como acessar os dados por um índice.

- Não aceita repetição, ou seja, não aceita objetos iguais, ou seja, que tenham os mesmos *hashCode* ou *equals*.

- Os dados podem ser heterogêneo, ***mas não é aconselhável***.

- Os dados ***podem e devem ser homogêneos*** - é melhor criar várias estruturas, cada uma com seu tipo de dado específico do que uma estrutura heterogênea contendo dados de mais de um tipo, por exemplo misturar *clientes*, *fornecedores* e *produto*.

**2) - List -** define uma coleção ordenada, podendo conter elementos duplicados. Em geral, o usuário tem controle total sobre a posição onde cada elemento é inserido e pode recuperá-los através de seus índices. Prefira esta interface quando precisar de acesso aleatório, através do índice do elemento.

- Características - Indexada - os dados podem ser acessados pelo índice.

- Aceita repetição

**3) - Map** - mapeia chaves para valores. Cada elemento tem na verdade dois objetos: uma chave e um valor. Valores podem ser duplicados, mas chaves não. SortedMap é uma interface que estende Map, e permite classificação ascendente das chaves. Uma aplicação dessa interface é a classe Properties, que é usada para persistir propriedades / configurações de um sistema, por exemplo.

- Características - É uma estrutura chave/valor como se fossem duas colunas de dados, por ex. *Nome/CPF.*

- Chave não aceita repetição

- Valor aceita repetição

**4) - Queue -** um tipo de coleção para manter uma lista de prioridades, onde a ordem dos seus elementos, definida pela implementação de Comparable ou Comparator, determina essa prioridade. Com a interface fila pode-se criar filas e pilhas.

- Características - Implementa uma FILA *(queue)*

- Tem a lógica *FiFo* - *First in / First out* - primeiro a entrar é o primeiro a sair.

**5) - Stack**

- Características - Implementa uma PILHA *(stack)*

- Tem a lógica *LiFo* - *Last in / First out* - último a entrar é o primeiro a sair.

**4) - MÉTODOS**

**4.1 - Convenções** - Uma das convernções sobre Métodos é a seguinte:

- método usa ***PARÂMETROS***

- o chamador do método usa ***ARGUMENTOS***.

- ex.: ***livro.setPaginas(236);*** - o objeto ‘livro’ chama o método setPaginas() passando o ***argumento*** 236.

- ex.: ***setPaginas(int nPag);*** - o método usa o ***parâmetro ‘int nPag’*** para receber o ***argumento*** 236.

**4.2 - Identificação do Parâmetro** -Neste caso um ***parâmetro*** nada mais é do que uma ***variável local*** identificada por um ***tipo*** e um ***nome*** que poderá ser usada dentro do corpo do método.

- É importante salientar que se um método usa parâmetro, então é preciso passar algo para ele *(pois ele sempre espera receber algo)* e este algo deverá ser um valor do tipo apropriado.

**4.2 - Tipo de Retorno** - Os métodos são sempre declarados ***precedidos por um tipo*** que será o ***retorno***, exceto o tipo ‘***void’*** que segnifica ‘***vazio’***, ou seja não retorna nada. Todos os ***tipos primitivos*** são considerados tipo de retorno efetivo e neste caso será exigido declarar no corpo do método o comando ‘return’.

- ex.: int setPaginas(int nPag){

return 239;}

**4.3 - Parâmetros** - Um método pode ter vários parâmetros, portanto você pode inviar mais de um argumento para ele, porém lembre-se que devem ser separados por vírgula e devem ser do tipo e na ordem correta em que os parâmetros foram declados no método.

**4.5 - Passagem de Argumentos** - Você pode passar valores ou variáveis contendo os valores para o método:

- ex.: nPaginas = 236;

nCapitulos = 35;

livro.setLivro(nPaginas, nCapitulos);

**4.6 - Valor ou Variável** - O valor ou a variável passados para um método são sempre uma cópia, isto significa que se estes dados forem alterados no corpo do método, os valores originais não são alterados.

**4.7 - Retorno** - Em um método pode-se declarar apenas um valor de retorno, quando é preciso retornar mais de um valor deve-se usar uma matriz (arrray).

**4.8 - Método Abstrato** - Um método Abstrato significa que ele deve ser sobreposto e não possui corpo, ou seja, não possui chave de abertura ( { ) e fechamento ( } ), simplesmente é finalizado com um ponto-e-vírgula.

- Exemplo: ***public abstract void comer();***

- Um método abstrato está inserido em uma classe abstrata, ou seja, NÃO É POSSÍVEL ter um método abstrato em uma classe NÃO abstrata. Porém é possível combinar métodos abstratos e NÃO abstratos em uma classe abstrata.

**4.9 - Método static** - Os métodos estáticos são chamados usando o nome da classe e não a variável de instância. Exemplo: Math.random()

**5) - Encapsulamento**

**6) - Classe Integer - É uma Wrapper Class - (faz parte da Biblioteca Java.lang que vem junto com o Java)**

6.1 - ***Integer.parseInt(“3”);*** - O método ***parseInt()*** da classe ***wrapper Integer*** converte uma string em um inteiro.

6.2 - ***Integer.toString(3);*** - O método ***toString()*** da classe ***wrapper Integer*** converte uma inteiro em uma string.

**7) - Loop for Aprimorado**

7.1 - Serve especificamente para iterar pelos elementos de uma matriz (array) ou outros conjuntos de dados.

7.2 - **for (int célula : celulaLocal) { }** - A variável **int celula** armazena um elemento da matriz **celulaLocal** até que não haja mais elementos ou seja introduzida uma interrupção **break**.

**8) - ARRAYLIST**

8.1) - **A ArrayList classe** - é um [**array**](https://www.w3schools.com/java/java_arrays.asp)**redimensionável** , que pode ser encontrado no pacote java.util.

8.2) - **Diferença entre array integrado e ArrayList** - A diferença entre um array integrado e um ArrayList em Java é que o tamanho de um array não pode ser modificado, (se você deseja adicionar ou remover elementos de/para um array, é necessário criar um novo). Enquanto os elementos podem ser adicionados e removidos de um ArrayList sempre que você quiser.

9) - **Sintaxe** - É também é um pouco diferente: Exemplo, crie um ArrayList objeto chamado **cars** que rmazenará strings:

*import java.util.ArrayList;*

*public class Main {*

*public static void main(String[] args) {*

*ArrayList <String> cars = new ArrayList<String>();*

*cars.add("Volvo");*

*cars.add("BMW");*

*cars.add("Ford");*

*cars.add("Mazda");*

*System.out.println(cars);*

*}*

*}*

**9) - API JAVA**

9.1) - **A sigla API** refere-se ao termo em inglês "**Application Programming Interface"** que significa em tradução para o português "**Interface de Programação de Aplicativos**".

9.2) - **Pacotes** - Todas as classes da biblioteca Java pertencem a um pacote. Por exemplo a classe ArrayList está no pacote chamado java.util, o qual contém uma pilha de classes utilitárias. O nome completo é **java.util.ArrayList**. Temos ainda muitas outras como o **java.net**, **java.io**, etc.

9.3) - **Como usar** - São duas opções: informar através da importação - import **java.util.ArrayList;** ou digitar o nome completo em todos os locais do código onde for usar - **java.util.ArrayList<Dog> lista = new java.util.ArrayList<Dog>;**

9.4) - **Pacote java.lang** - Existem, porém várias classes que estão dentro do pacote java.lang e não precisam ser importadas ou declaradas, pois fazerm parte da biblioteca padrão do java, por exemplo:

**- java.lang.System.out**

**- java.lang.Math**

**- java.lang.String**

**10) - Herança e Níveis de Acesso**

- ***Herança*** permite o reuso de ***atributos*** e ***métodos*** (dados e comportamento). É um tipo de associação que permite que uma classe herde todos dados e comportamentos de outra classe.

- Quando uma classe herda de outras, dizemos que a ***subclasse estende*** (*extends*) a superclasse. Para saber se uma classe deve estender outra, aplique o teste E-UM:

- A ***classe Triângulo*** É-UMA subclasse da ***superclasse Forma***, isso faz sentido. Mas Forma não É-UMA subclasse de Triângulo pois não faz sentido.

- A classe Gato É-UMA subclasse da superclasse Felino, isso também faz sentido, mas o inverso não.

- É preciso lembrar que o relacionamento É-UM funciona somente em uma direção, pois não existe herança inversa ou regressiva, por exemplo: as crianças herdam dos pais e não o contrário.

- Um método da superclasse pode ser usado junto com o correspondente método da subclasse o qual vai somente adicionar algo a ele. O modo de fazer isso é usar a palavra chave ***super*** no método de sobreposição da subclasse para chamar aquele da superclasse. É como dizer “primeiro execute a versão da superclasse e, em seguida volte e termine com o meu próprio código.”

- Não se usa a herança objetivando apenas reutilizar o código de outra classe se o relacionamento entre a subclasse e a superclasse violar a regra do É-UM.

***10.1 - Níveis de Acesso***

- Há 03 níveis de acessso a serem abordados. Os níveis de acesso controlam quem vê o quê e são essenciais em um código Java robusto e bem projetado.

- Os símbolos entre parêntesis ***(+), (-) e (#)*** são aqueles usadas nos diagramas da UML para indicar o nível de acesso das Classes, Métodos e Atributos.

- ***public (+)*** (público) - Membros públicos PODEM ser herdados. Os membros ***public*** de uma classe são acessíveis onde quer que o programa tenha uma referência a um objeto dessa classe ou a uma de suas subclasses.

- ***private (-)*** (privado) - Membros privados NÃO podem ser herdados. Podem ser acessados apenas dentro da classe. Membros private de uma superclasse não são acessíveis fora da própria classe. Em vez disso, eles permanecem ocultos de suas subclasses e só podem ser ***acessados*** por meio dos ***métodos public*** ou ***protected*** herdados da superclasse.

- ***protected (#)*** (protegido) - Os membros private de uma classe só são acessíveis dentro da própria classe. Utilizar acesso ***protected*** oferece um ***nível intermediário*** de acesso entre ***public*** e ***private***.

- Os membros ***protected*** de uma superclasse podem ser acessados por membros dessa superclasse, de suas subclasses e de outras classes no mesmo pacote — membros protected também têm acesso de pacote.

***10.2 - Casting***

- ***Upcasting***

• Casting da subclasse para superclasse

• Uso comum: polimorfismo

- ***Downcasting***

• Casting da superclasse para subclasse

• Palavra instanceof

• Uso comum: métodos que recebem parâmetros genéricos (ex: Equals)

***10.3 - Sobreposição ou sobrescrita***

- É a implementação de um método de uma superclasse na subclasse.

- É fortemente recomendável usar a anotação @Override em um método sobrescrito pois facilita a leitura e compreensão do código avisamos ao compilador (boa prática).

***10.4 - Palavra reservada Super***

- É possível chamar a implementação da superclasse usando a palavra super.

- Um método da superclasse pode ser usado junto com o correspondente método da subclasse o qual vai somente adicionar algo a ele. O modo de fazer isso é usar a palavra chave ***super*** no método de sobreposição da subclasse para chamar aquele da superclasse. É como dizer “primeiro execute a versão da superclasse e, em seguida volte e termine com o meu próprio código.”

- Exemplo: suponha que, na classe BusinessAccount, a regra para saque seja realizar o saque normalmente da superclasse, e descontar mais 2.0.

*@Override*

*public void withdraw(double amount) {*

*super.withdraw(amount);*

*balance -= 2.0;*

*}*

***10.5 - Palavra chave Final em Classes e Métodos***

A palavra chave final evita que classes e métodos sejam herdados. Exemplo:

**Classe**:

*public* ***final*** *class SavingsAccount {*

**Método**:

*public* ***final*** *void withdraw(double amount) {*

*balance -= amount;*

*}*

***Pra quê?***

• **Segurança**: dependendo das regras do negócio, às vezes é desejável garantir que uma classe não seja herdada, ou que um método não seja sobreposto. Geralmente convém acrescentar ***final*** em ***métodos sobrepostos***, pois sobreposições múltiplas podem ser uma porta de entrada para inconsistências

• **Performance**: atributos de tipo de uma classe final são analisados de forma mais rápida em tempo de execução.

• Exemplo clássico: ***Classe String*** *(*[*https://docs.oracle.com/javase/7/docs/api/java/lang/String.html*](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) *)*

**11) - Formatação de Números - Método String.format() e System.out.printf()**

1 - A formatação é composta de duas partes separadas por vírgula *(isso vale tanto para String.format() como para System.out.printf().)*:

- **System.out.printf("%,d", 100000000);**

- **String s = String.format("%,d", 100000000);**.

- primeira parte entre aspas duplas contém o sinal de **%** que indica onde o **argumento** será posicionado, e **a vírgula + a letra ‘d’** insere pontos na formatação do número.

- segunda parte contém o argumento a ser formatado, no caso o número 100000000.

2 - Pode-se formatar mais de um argumento: **System.out.printf("%,d %,f", 100000000, 45478.23167);**

3 - Usa-se **a letra ‘d’** para formatar números inteiros decimais e a **letra ‘f’** para números de ponto flutuante.

4 - **“%,d”** 🡺 [System.out.printf("%,d", 100000000);] - formata String '100000000' com pontos = **100.000.000**

5 - **“%n”** 🡺 [System.out.printf("%n");] -salta linha (pode ser usado junto com outras formatações.

6 - **“%.2f”** 🡺 [System.out.printf("%.2f", 47.09876);] *-* especifica a precisão decimal de '47.09876' = **47,10**

7 - **“%,.2f”** 🡺 [System.out.printf("%,.2f”, 346768.496578);] - formata String '346768.496578' e a precisão decimal = **346.768,50**

8 - **“%x”** 🡺 [System.out.printf("%x", 42);] - retorna o valor hexadecimal de '42' = **2a**

9 - **“%c” 🡺** [System.out.printf("%c, 42);] - retorna o caractere do número '42' = **\***

**12) - Formatação de Datas**

1 - **%tc** 🡺 System.out.printf(“Data atual do sistema = %tc %n", new Date());

2 - **%tr** 🡺 System.out.printf("Somente Hora = %tr", new Date());

3 - **%tA, %tB %td** 🡺 System.out.printf("Dia semana, mês e dia mês = %tA, %tB %td", new Date(),new Date(), new Date());

4 - %**tA, %<tB %<td** 🡺 System.out.printf("Dia semana, mês e dia mês = %tA, %<tB %<td", new Date()); Use sinal '<' para não repetir o new Date().

**13) - GUI - Graphic User Interface**

- Um JFrame é um objeto que representa uma janela na tela.

13.1 - Como Criar uma GUI

1 - Crie uma moldura:

***JFrame frame = new JFrame();***

2 - Crie um elemento gráfico (botão, campo de texto, etc.):

***JButton button = new JButton(“click me”);***

3 - Adicione o elemento gráfico à moldura:

***frame.getContentPane().add(Button);***

4 - Exiba a GUI fornecendo um tamanho de torne-a visível:

***frame.setSize(300,300);***

***frame.setVisible(true);***

**14) - Anotação @Override e o Método toString**

- O ***método toString*** pertence à ***classe Object*** e pode ser ***sobrescrito*** (override) por qualquer subclasse.

- O toString (linhas 93 a 101) é especial — é um dos métodos que toda classe herda direta ou indiretamente da classe Object (resumida na Seção 9.6). O método toString retorna uma String que representa um objeto. Este método é chamado implicitamente sempre que um objeto deve ser convertido em uma representação String, como quando um objeto é impresso pelo método printf ou por método String format via o especificador de formato %. O método toString da classe Object retorna uma String que inclui o nome da classe do objeto. Este método é principalmente um marcador de lugar que pode ser sobrescrito por uma subclasse para especificar uma adequada representação String dos dados em um objeto de subclasse.

- Quando um objeto (employee) é enviado para a saída (System.out), utilizando o especificador de formato %s, o ***método toString*** do objeto é invocado implicitamente para obter a representação da String do objeto.

- *System.out.printf("%n%s%n%n%s%n", "Atualiza informações obtidas via toString", employee);*

**15) - Classe Enumeração e Composição**

- É um tipo especial que serve para especificar de forma literal um conjunto de constantes relacionadas.

- São tipos de campos que consistem em um conjunto fixo de constantes (static final), como uma lista de valores pré-definidos. Na linguagem de programação Java, pode ser definido um tipo de enumeração usando a palavra chave ***enum***.

***• Referência:*** <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

- A ***classe enum*** possui vários ***MÉTODOS***. Mais informações acesse o linkda documentação onde irá encontrar mais características e métodos <https://docs.oracle.com/javase/6/docs/api/java/lang/Enum.html>

- Todos os ***tipos enums*** implicitamente estendem a ***classe java.lang.Enum***, sendo que o Java não suporta herança múltipla, não podendo estender nenhuma outra classe.

- **Em relação às propriedades é preciso tomar os seguintes cuidados:**

* As instâncias dos tipos enum são criadas e nomeadas junto com a declaração da classe, sendo fixas e imutáveis (o valor é fixo).;
* Não é permitido criar novas instâncias com a palavra chave new;
* O construtor é declarado private, embora não precise de modificador private explícito;
* Seguindo a convenção, por serem objetos constantes e imutáveis (static final), os nomes declarados recebem todas as letras em MAIÚSCULAS;
* As instâncias dos tipos enum devem obrigatoriamente ter apenas um nome;
* Opcionalmente, a declaração da classe pode incluir variáveis de instância, construtor, métodos de instância, de classe, etc.

- **Veja exemplos de código nos Projetos:**

*D:\CursosUdemy\CursoJavaNelioAlves\Projetos\_NetBeans\Ex13\_115\_Enums*

*D:\CursosUdemy\CursoJavaNelioAlves\Projetos\_NetBeans\Ex13\_115-1\_Enums*

**16) - Classe StringBuilder**

- A classe ***StringBuilder*** faz parte do pacote ***java.lang***. Essa classe permite criar e manipular dados de Strings dinamicamente, ou seja, podem criar variáveis de String modificáveis.

- Uma vantagem sobre a ***classe String*** é a concatenação de strings. Pois quando concatena strings com StringBuilder é invocado o ***método append***. Esse método aloca novas strings concatenadas para o mesmo objeto, ou seja, cada vez que concatena strings não são criadas cópias dos objetos como é realizado pelo ***método concat*** da ***classe String***, contribuindo para um melhor desempenho do sistema.

**- Codigo Exemplo:**

***package testastringbuilder;***

***public class Ex13\_120\_TestaStringBuilder {***

***public static void main(String[] args) {***

***//Instancia um objeto 'nome' do tipo StringBuilder.***

***StringBuilder nomes1 = new StringBuilder();***

***StringBuilder nomes2 = new StringBuilder();***

***//EXIBE os nomes na mesma linha.***

***nomes1.append("Carlos - ").append("Maria - ").append("José - ").append("Renata\n");***

***//EXIBE um nome por linha usando instrução '\n'.***

***nomes1.append("Rafael - \n").append("Ana - \n").append("Manoel - \n").append("Rocco\n");***

***//EXIBE os nomes com um 'append' e '\n' por linha.***

***nomes2.append("BRENO\n");***

***nomes2.append("DIANA\n");***

***nomes2.append("JOÃO\n");***

***nomes2.append("JUSTIN\n");***

***System.out.println(nomes1.toString() + nomes2.toString());***

***}***

***}***

- Veja outro exemplo de código no Projeto *D:\CursosUdemy\CursoJavaNelioAlves\ws\_Eclipse\Ex13\_120\_ComposicaoStringBuilder*

**17) - Polimorfismo**

- Contar com o fato de que cada objeto sabe “fazer a coisa certa” *(isto é, faz o que é apropriado a esse tipo de objeto)* em resposta à mesma chamada de método é o conceito-chave do polimorfismo. A mesma mensagem (nesse caso, mover) enviada a uma variedade de objetos tem muitas formas de resultados — daí o termo polimorfismo.

**18) - Classes Abstratas**

- São ***classes*** que não podem ser ***instanciadas***.

- É uma forma de garantir ***herança*** total: somente ***subclasses*** **não** abstratas podem ser instanciadas, mas nunca a ***superclasse abstrata.*** - Sintaxe: *public* ***abstract*** *class Account { (...) }*

- Vantagens de utilizar ***Classes Abstratas:***

**- Reuso**

- ***Polimorfismo***: a superclasse classe genérica nos permite tratar de forma fácil e uniforme todos os tipos de conta, inclusive com polimorfismo se for o caso (como fizemos nos últimos exercícios). Por exemplo, você pode colocar todos tipos de contas em uma mesma coleção.

- Demo: suponha que você queira:

- Totalizar o saldo de todas as contas.

- Depositar 10.00 em todas as contas

Nota: para as outras classes ver o projeto completo no NetBeans.

*package heranca\_Main;*

*import entities.Account;*

*import entities.BusinessAccount;*

*import entities.SavingsAccount;*

*import java.util.ArrayList;*

*import java.util.List;*

*import java.util.Locale;*

*public class Ex14\_125\_Heranca {*

*public static void main(String[] args) {*

*Locale.setDefault(Locale.US);*

*//Cria uma lista do tipo 'List' para receber dados das contas.*

*List<Account> list = new ArrayList<>();*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*//Instancia as variáveis de instância de todoas as 04 contas.*

*list.add(new SavingsAccount(1001, "Alex", 500.00, 0.01));*

*list.add(new BusinessAccount(1002, "Maria", 1000.0, 400.0));*

*list.add(new SavingsAccount(1004, "Bob", 300.0, 0.01));*

*list.add(new BusinessAccount(1005, "Anna", 500.0, 500.0));*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*//Exibe o saldo de todas as contas.*

*double sum = 0.0;*

*for (Account acc : list) {*

*sum += acc.getBalance();*

*}*

*System.out.printf("Total balance: %.2f%n", sum);*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*//Efetua um depósito de 10.0 em cada conta.*

*for (Account acc : list) {*

*acc.deposit(10.0);*

*}*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*//Exibe o saldo atualizado de todas as contas com o depósito de 10.0.*

*for (Account acc : list) {*

*System.out.printf("Updated balance for account %d: %.2f%n", acc.getNumber(), acc.getBalance());*

*}*

*}*

*}*

- No diagrama da notação ***UML*** o nome da ***classe*** é escrito em ***itálico.***

**19) - Métodos Abstratos**

**20) - Métodos Vários**

- **toLowerCase()**

**- charAt(0);**

*System.out.print("Escolha Retângulo ou Círculo (r/c)? ");*

*char ch = sc.next().toLowerCase().charAt(0);*

**- valueOf(sc.next().**

**- toUpperCase());**

*System.out.print("Escolha a cor da forma (BLACK/BLUE/RED): ");*

*CorForma cor\_ForGeom = CorForma.valueOf(sc.next().toUpperCase());*

**- split()** - O método split quebra uma String em várias substrings a partir de um caracter definido por você.

**- Exemplo-1**

*string texto = “banana,maçã,laranja”;*

*stringfrutas [] = texto.split(",");*

*System.out.println(frutas [0]); //imprime banana*

*System.out.println(frutas [1]); //imprime maçã*

*System.out.println(frutas [2]); //imprime laranja*

**- Exemplo-2**

- *String[] nomes = sc.nextLine().split(" ");* 🡺 Lê os nomes via teclado e os armazena em um array separados por espaço em branco.

**21) - Tratamento de Exceções**

- Hierarquia de exceções do Java *(*[*https://docs.oracle.com/javase/10/docs/api/java/lang/package-tree.html*](https://docs.oracle.com/javase/10/docs/api/java/lang/package-tree.html)*)*

***-Throwable*** *🡺 Super Classe genérica contendo os erros e as exceções (abaixo alguns error e exceptions)*

***- Classe Error*** *🡺 Contém os erros que o programa não consegue tratar.*

*- OutOfMemoryError*

*- VirtualMachineError*

***- Classe Exception*** *🡺 Contém as exceções que o programa deve tratar.*

*- IOException*

*- RuntimeException*

*- IndexOutOfBoundsException*

*- NullPointerException*

**21.1 -** **Estrutura try - catch** - *(try = tente executar - catch = capture a excessão, ou seja, o erro) - E*vita que o programa seja interrompido subitamente por um erro inesperado tipo falta de memória, índice array fora do limite ((ArrayIndexOutOfBoundsException), incompatibilidade de entrada (InputMismatchException), etc.

- ***Bloco try*** - Contém o código que representa a execução normal do trecho de código que pode acarretar em uma exceção.

- ***Bloco catch*** - Contém o código a ser executado caso uma exceção ocorra. Deve ser especificado o tipo da exceção a ser tratada *(upcasting é permitido).*

*try {*

*<aqui entra o bloco de código que pode gerar uma ou mais exceções>*

*} catch (ExceptionType e) {*

*<aqui entra o código a executar caso ocorra a excessão>*

*} catch (ExceptionType e) {*

*<aqui entra o código a executar caso ocorra a excessão>*

*} catch (ExceptionType e) {*

*<aqui entra o código a executar caso ocorra a excessão>*

*}*

**21.2 - Método printStackTrace()** - *Pilha de chamadas de métodos* - Adicionando o método ***printStackTrace()*** dentro do ***bloco catch***, o sistema exibe o erro e a ***pilha de métodos chamadores,*** facilitando a compreensão do mesmo.

**21.3 - Bloco finally** - É um bloco que contém código a ser executado ***independentemente*** de ter ocorrido ou não uma exceção. Exemplos clássicos: *fechar um arquivo, conexão de banco de dados*, ou outro recurso específico ao final do processamento.

**- Sintaxe:**

*try {*

*}catch (ExceptionType e) {*

*} finally {*

*}*

**Exemplo de try - catch - printStackTrace():**

*/\**

*\*/*

*package application;*

*import java.util.InputMismatchException;*

*import java.util.Scanner;*

*public class Programa\_Principal\_140 { //Início da classe*

*public static void main(String[] args) {*

*metodo1();*

*System.out.println("FIM DO PROGRAMA!");*

*}*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*public static void metodo1() {*

*System.out.println("INÍCIO METODO1");*

*metodo2();*

*System.out.println("FIM METODO1");*

*}*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*public static void metodo2() {*

*System.out.println("INÍCIO METODO2");*

*//Declara uma classe Scanner p/ capturar dados do teclado.*

*Scanner sc = new Scanner(System.in);*

*int copiaIndice = 0;*

*//Inicio do bloco try-catch.*

***try {***

*System.out.print("Digite os nomes: ");*

*String[] nome = sc.nextLine().split(" ");*

*copiaIndice = nome.length - 1;*

*System.out.print("Digite o número do índice: ");*

*int indiceNome = sc.nextInt();*

*System.out.println("Nome[" + indiceNome + "] = " + nome[indiceNome]);*

***} catch*** *(ArrayIndexOutOfBoundsException e) {*

*System.out.println("Número digitado tem que ser de 0 a " + copiaIndice);*

***e.printStackTrace(); //Exib telae a excessão na tela.***

***} catch*** *(InputMismatchException e) {*

*System.out.println("Digite somente número inteiro.");*

***e.printStackTrace(); //Exib telae a excessão na tela.***

***}***

*System.out.println("FIM METODO2");*

*}*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*}//Fim da classe.}*

**Exemplo de finally:**

*/\*Projeto - Ex15\_141\_BlocoFinally*

*\*/*

*package application;*

*import java.io.File;*

*import java.io.IOException;*

*import java.util.Scanner;*

*public class Programa\_Principal\_141 {*

*public static void main(String[] args) {*

*//Classe File - instancia um arquivo do tipo 'File' recebendo um arquivo texto.*

*File arqTexto = new File("D:\\Temporarios\\Ex15\_141\_BlocoFinally.txt");*

*Scanner sc = null;*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*//Blocos try-catch-finally*

***try {***

*sc = new Scanner(arqTexto);*

*while (sc.hasNextLine()) {*

*System.out.println(sc.nextLine());*

***}***

***} catch (IOException e) {***

*System.out.println("ERRO AO ABRIR O ARQUIVO: " + e.getMessage());*

***} finally {***

*if (sc != null) {*

*sc.close();*

***}***

*System.out.println("Bloco 'finally' executado!!!");*

*}*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*}*

*}//Fim da classe*

**22) - Classes Várias**

**1 - Classe File** - Representação abstrata de um arquivo e seu caminho.

Pertence ao pacote ***java.io.File.*** Instancia uma variável do tipo ***File*** para receber um arquivo texto.

- Sintaxe: ***File arqTexto = new File("D:\\Temporarios\\Ex15\_141\_BlocoFinally.txt");***

***- (Exemplo de código: Ex17\_183\_File\_Scanner\_Classes)***

- Documentação Java - <https://docs.oracle.com/javase/10/docs/api/java/io/File.html>

**2 - Classe Scanner** - Leitor de texto. Pertence ao pacote ***java.util.Scanner***. Instancia uma variável do tipo ***Scanner*** que lê os dados digitados através de:

***1 - teclado -*** Sintaxe: ***Scanner sc = new Scanner(System.in);***

***2 - arquivo*** - Sintaxe: ***File arquivoTeste = new File ("D:\\Temporarios\\reflexoes\_1.txt");***

- *(Nota: são colocadas duas barras \\ porque a primeira barra é o prefixo de caracteres especiais.)*

***- (Exemplo de código: Ex17\_183\_File\_Scanner\_Classes)***

- Utiliza o método ***hasNextLine()*** para verificar se existe uma próxima linha a ser lida.

- Utiliza o método ***nextLine()*** para ler as linhas texto do arquivo.

- Sintaxe: ***while (sc.hasNextLine()) { System.out.println(sc.nextLine()); }***

***- (Exemplo de código: Ex17\_183\_File\_Scanner\_Classes)***

- Documentação Java - <https://docs.oracle.com/javase/10/docs/api/java/util/Scanner.html>

**3 - Classe IOException** *(extends Exception)* - Pertence ao pacote ***java.io.IOException*** que é a exceção padrão para ***entrada*** e ***saída*** de arquivos. É uma classe que herda da ***classe Exception*** a qual obriga a tratar as exceções.

- Sintaxe: ***catch (IOException e) { System.out.println ("ERRO: " + e.getMessage()); }***

- Quando o arquivo não é encontrado é gerada automaticamente a msg:

*D:\Temporarios2\reflexoes\_1.txt (O sistema não pode encontrar o caminho especificado)*

***- (Exemplo de código: Ex17\_183\_File\_Scanner\_Classes)***

- Documentação Java - <https://docs.oracle.com/javase/10/docs/api/java/io/IOException.html>

**4 - Classe FileReader** - ***Stream*** *(em programação significa uma sequência)* de leitura de caracteres a partir de arquivos.

- Sintaxe: *public static void main(String[] args) {*

*String path = "D:\\Temporarios\\reflexoes\_1.txt";*

*FileReader fr = null;*

*BufferedReader br = null;*

*try {*

*//br = new BufferedReader(new FileReader(path));*

*fr = new FileReader(path);*

*br = new BufferedReader(fr);*

*String line = br.readLine();*

*while (line != null) {*

*System.out.println(line);*

*line = br.readLine();*

*}*

*}*

*}*

***- (Exemplo de código: Ex17\_184\_FileReader\_BufferedReader***

- Documentação Java - <https://docs.oracle.com/javase/10/docs/api/java/io/FileReader.html>

**5 - Classe BufferedReader** - Junto com o FileReader, agiliza a leitura do arquivo.

- Sintaxe: *(ver FileReader)*

***- Exemplo de código: Ex17\_184\_FileReader\_BufferedReader***

- Documentação Java - <https://docs.oracle.com/javase/10/docs/api/java/io/BufferedReader.html>

**6 - Bloco try-with-resources** - É um bloco try que declara um ou mais recursos, e garante que esses recursos serão fechados ao final do bloco.

- Portanto não é preciso abrir um bloco finally para garantir que os recursos serão fechados como é feito para as classes ***File, Scanner, FileReader e BufferedReader***.

- Sintaxe: *public static void main(String[] args) {*

*String path = "D:\\Temporarios\\reflexoes\_1.txt";*

*try ( BufferedReader br = new BufferedReader(new FileReader(path))) {*

*String line = br.readLine();*

*while (line != null) {*

*System.out.println(line);*

*line = br.readLine();*

*}*

*} catch (IOException e) {*

*System.out.println("ERROR: " + e.getMessage());*

*}*

*}//Fecha main*

***- Exemplo de código: Ex17\_185\_ try-with-resources***

- Documentação Java - <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

**7 - FileWriter e BufferedWriter -** Cria um arquivo ou sobrescreve um existente. Para adicionar texto ao final de um arquivo existente usa-se o valor *true*. Stream de escrita de caracteres em arquivos.

**-** Cria/recria o arquivo: new FileWriter(path).

**-** Acrescenta ao arquivo existente: new FileWriter(path, true).

***- Sintaxe: try ( BufferedWriter bw = new BufferedWriter(new FileWriter(path))) {} -***

***try ( BufferedWriter bw = new BufferedWriter(new FileWriter(path\_2, true))) {}***

***- Exemplo de código:* *Ex17\_186\_ FileWriter\_BufferedWriter***

- Documentação Java - <https://docs.oracle.com/javase/10/docs/api/java/io/FileWriter.html>

**8 - Manipulando pastas com File**

- Listar as ***pastas*** de um diretório:

***Sintaxe: String strCaminho = sc.nextLine(); //obtido via teclado***

***File caminho = new File(strCaminho);***

***//Cria uma variável ‘pastas’ do tipo vetor de File.***

***File[] pastas = caminho.listFiles(File::isDirectory); //Esta é uma sintaxe chamada de Reference Methods (Referência a métodos) - File::isDirectory lista somente pastas.***

- Listar os ***arquivos*** de um diretório:

***Sintaxe: String strCaminho = sc.nextLine(); //obtido via teclado***

***File caminho = new File(strCaminho);***

***//Cria uma variável ‘arquivos’ do tipo vetor de File.***

***File[] arquivos = caminho.listFiles(File::isFile); //File::isFile lista somente os arquivos.***

***- Exemplo de código:* *Ex17\_187\_ File\_ManipulandoPastas***

- Documentação Java - ???

**9 - Informações do caminho do arquivo** - Manipulações várias utilizando algumas funções *(existem muitas outras)* para o caminho digitado. Estas funções NÃO verificam se o caminho digitado é verdadeiro, apenas manipula a String digitada:

**- *getPath()*** - retorna o caminho completo

**-** ***getParent()*** - retorna o caminho sem o arquivo

**-** ***getName()*** - retorna somente nome do arquivo

***Sintaxe: String strCaminho = sc.nextLine();***

***File caminho = new File(strCaminho);***

***System.out.println("Caminho completo: " + caminho.getPath());***

***System.out.println("Caminho sem o arquivo: " + caminho.getParent());***

***System.out.println("Somente nome do arquivo: " + caminho.getName());***

***sc.close();***

**10 - Classe SimpleDateFormat** - Pertence ao pacote ***java.text.SimpleDateFormat***. Instancia uma variável do tipo SimpleDateFormat que recebe o formato da máscara da data.

- Sintaxe: ***SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");***

**23) - Interface**

- Interface é um tipo de classe que define um conjunto de operações que uma outra classe deve implementar.

- Pra quê interfaces? Para criar sistemas com baixo acoplamento e flexíveis.

- Um ***acoplamento forte*** e pouco flexível acontece, quando é feita uma associação direta de uma classe para outra, criando uma dependência, por não se usar uma interface. Isso significa que em caso de necessidade de manutenção será preciso modificar várias classes.

- A ***interface*** estabelece um ***contrato*** com as classes que a ***implementam***, ou seja, estas classes são obrigadas a implementar todos os ***métodos*** especificados pela ***interface***.

**1 - Inversão de controle** - Padrão de desenvolvimento que consiste em retirar da classe a responsabilidade de instanciar suas dependências através do ***construtor***, o que iria gerar o mesmo ***forte acoplamento***. Ao invés disso se usa a ***Injeção de dependência.***

**2 - Injeção de dependência** - uma forma de realizar a inversão de controle: um componente externo *(poderia ser o programa principal)* instancia a dependência, que é então injetada no objeto "pai".

- Código Exemplo usando a ***Injeção de dependência***:

***class ProgramaPrincipal {***

***public static void Main(string[] args) {***

***(...)***

***//Instancia o objeto ‘rentalService’.***

***RentalService rentalService = new RentalService(pricePerHour, pricePerDay, new BrazilTaxService());***

***}***

***}***

***------------------------------------------------------------------------------------------------------------------------------------------***

***- O objeto pai referencia a interface:***

***class RentalService {***

***private TaxService taxService;***

***//Construtor***

***public RentalService (double pricePerHour, double pricePerDay, TaxService taxService) {***

***this.pricePerHour = pricePerHour;***

***this.pricePerDay = pricePerDay;***

***this.taxService = taxService;***

***}***

***}***

- Código Exemplo de como seria uma ***NÃO*** ***Injeção de dependência****:*

***class RentalService {***

***//Atributo do tipo interface 'TaxService', associado a RentalService***

***private TaxService taxService;***

***//Construtor***

***public RentalService (double pricePerHour, double pricePerDay) {***

***this.pricePerHour = pricePerHour;***

***this.pricePerDay = pricePerDay;***

***this.taxService = new BrazilTaxService();***

- A ***Injeção de dependência*** pode ser implementada de várias formas:

- Construtor

- Classe de instanciação (builder / factory)

- Container / framework

**3 - Herança vs Interface** -

**- Possuem alguns aspectos em comum:**

***- Relação é-um*** - Na ***herança*** temos ***subclasses*** herdando de uma ***superclasse*** ***abstrata*** e na ***Interface*** temos ***classes*** ***concretas*** implementando ***funções*** desta ***interface***.

***- Generalização/especialização*** - Na ***herança*** a ***superclasse*** é um ***tipo*** ***genérico*** e as ***subclasses*** são a especialização e do mesmo modo a ***Interface*** é um tipo ***genérico*** e as ***classes*** ***concretas*** são a especialização.

***- Polimorfismo*** - Se tivermos uma ***variável*** do ***tipo*** ***classe*** ***abstrata*** (ex.: classe FormaGeometrica), em tempo de execução esta variável pode ser ***associada*** a qualquer ***subclasse*** que extende a ***classe*** ***abstrata*** (ex.: Circulo, Triangulo, etc.) e uma operação qualquer (ex.: método área()) vai se comportar (sobrescrever), conforme a especificidade da ***subclasse***. Da mesma forma se tivermos uma ***variável*** do ***tipo*** ***Interface*** ela vai se comportar conforme a especificidade da ***classe*** ***concreta***.

**- Possuem também diferenças fundamentais**

***- Herança =>*** reuso de ***informações*** *(ex.: atributo Color CorForma)* e ***comportamentos*** *(ex.: métodos getters/setters).*

***- Interface =>*** o objetivo primordial é um ***contrato a ser cumprido,*** ou seja, qualquer classe concreta que implementa a Interface é obrigada a implementar seus métodos.

***- Exemplo de código: Projeto Ex18\_199\_Heranca\_Com\_Interface***

**4 - Herança múltipla e o problema do diamante** - A ***herança múltipla*** pode gerar o ***problema do diamante***: uma ***ambiguidade*** causada pela existência do mesmo ***método*** em mais de uma ***superclasse***. ***Herança múltipla*** não é permitida no JAVA e na maioria das linguagens!

- Porém, uma ***classe*** pode ***implementar*** mais de uma ***interface*** e ***extender*** uma classe ao mesmo tempo, mas ATENÇÃO: Isso NÃO é ***herança múltipla***, pois NÃO HÁ REUSO na relação entre a ***classe*** ***concreta*** *(no ex. a classe ComboDevice),* e as ***interfaces*** *(no ex. as interfaces Scanner e Printer)*. A ***classe*** ***concreta*** *(ComboDevide)* não herda, mas sim ***implementa*** as ***interfaces*** *(ou seja cumpre o contrato)*.

- Ex. sintaxe: *public class ComboDevice* ***extends*** *Device* ***implements*** *Scanner, Printer { }*

***- Exemplo de código: Projeto Ex18\_200\_HerancaMultipla\_ProblemaDoDiamante***

***- Apostila: D:\Cursos\_PROGRAMACAO\Cursos\_Udemy\Curso\_Java\_POOCompleto\_NelioAlves\ Material\_de\_Apoio \ Apostilas\_do\_curso\18\_191-interfaces.pdf***

**5 - Interface Comparable** - Deve ser usado com ‘List’ cujo tipo seja uma classe objeto e não String:

Existem duas interfaces para configurar dados em listas, a ***interface java.lang.Comparable*** e a ***interface java.util.Comparator***.

- Ex.: *public static void main(String[] args) {*

*List<Funcionario> listaNomes = new ArrayList<>();*

*}*

*public class Funcionario implements Comparable<Funcionario> {*

*@Override*

*public int compareTo(Funcionario outro) {*

*return nomeFunc.compareTo(outro.getNomeFunc()); //Retorna os nomes em ordem alfabética crescente.*

*}*

***- Exemplo de código: Projeto Ex18\_201\_Interface\_Com\_Comparable***

- https://docs.oracle.com/javase/10/docs/api/java/lang/Comparable.html

public interface Comparable {

int compareTo (T o); *//Declara um método abstrato ‘compareTo’ que recebe um objeto ‘o’ do tipo ‘T’.*

}

**6 - Default methods ou métodos padrão** *(também chamados de defender methods)*

- A partir do Java 8, ***interfaces*** podem conter ***métodos concretos***. Vale ressaltar que as interfaces podem também ter ***métodos*** ***estáticos*** e ***métodos*** ***privados***.

- A intenção básica é prover implementação padrão para métodos, de modo a evitar:

- 1) - repetição de implementação em toda classe que implemente a interface

- 2) - a necessidade de se criar classes abstratas para prover reuso da implementação

- Outras vantagens:

1) - Manter a retrocompatibilidade com sistemas existentes

2) - Permitir que "interfaces funcionais" *(que devem conter apenas um método,)* possam prover outras operações padrão reutilizáveis

- **Considerações importantes**:

- Sim, a partir do Java 8 as interfaces podem prover reuso

- Sim: agora temos uma forma de herança múltipla

- Mas o compilador reclama se houver mais de um método com a mesma assinatura, obrigando a sobrescreve-lo

- ***Interfaces*** ainda são bem diferentes de classes abstratas. Interfaces ***não*** ***possuem*** recursos tais como ***construtores*** e ***atributos***

***- Exemplo de código: Projeto Ex18\_202\_Default\_metodos***

**24) - Generics, Set e Maps**

**1 - Generics**

- Generics permitem que ***classes***, ***interfaces*** e ***métodos*** possam ser ***parametrizados*** por ***tipo***.

- Seus benefícios são:

*- Reuso*

*- Type safety*

*- Performance*

**- Uso comum: coleções**

Exemplo:

- *List<String> Rel\_nomes = new ArrayList<>();* 🡺 Parametriza a variável *Rel\_nomes* de uma ***coleção*** do tipo ***List*** para receber somente ***strings***.

- *Rel\_nomes.add("Maria");* 🡺 Adiciona a string ‘*Maria’* à variável *Rel\_nomes.*

*- String nome = Rel\_nomes.get(0);* 🡺 A variável *nome* do tipo *String* recebe o nome ‘*Maria’* que está na posição zero da variável *Rel\_nomes* invocando o método *get(0);* da *coleção* *List*.

***- Exemplo de código: Projeto Ex19\_205\_3Com\_Generics\_Sem\_Objects-3-Boa***

**25) - Programação Funcional e Expressões Lambda**

**1 - Cálculo Lambda**

- É o formalismo matemático, base da programação funcional.

**2 - Expressão lambda**

- É a Função Anônima de primeira classe.

**3 - Interface funcional Predicate**

- É uma interface que possui um único método abstrato. Suas implementações serão tratadas como expressões lambda.

- Existem no Java várias interfaces funcionais, abaixo algumas delas:

- ***Predicate*** - (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>)

- Exemplo-1: Uso de *predicate* direto no ***método*** ***removeIf()*** *(sem interface funcional)*:

- *//Instancia uma variável 'list' tipo 'List' parametrizada como 'Product'*

*List<Product> list = new ArrayList<>();*

*//Adiciona dados ao ArrayList 'list'.*

*list.add(new Product("TV", 900.00));*

*list.add(new Product("Mouse", 50.00));*

***list.removeIf(p -> p.getPrice() >= 100))***

- ***Exemplo de código: Projeto Ex20\_240\_ Predicate\_SEM\_InterfaceFuncional***

- Exemplo-2: Uso de *predicate* instanciando uma *interface* *funcional* no ***método removeIf()***

*//Instanciando uma interface funcional contendo o predicate*

***- list.removeIf(new ProductPredicate()));***

- ***Exemplo de código: Projeto: Ex20\_240\_2\_Predicate\_COM\_InterfaceFuncional***

- *Function* - (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>)

- *Consumer* (<https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html>)

- **Nota**: ao contrário das outras interfaces funcionais, no caso do Consumer, é esperado ele possa gerar efeitos colaterais, porque o método abstrato da interface consumer é um método *‘void’*, ou seja, ele simplesmente faz uma ação que pode ser qualquer coisa, tipo imprimir na tela ou alterar algo que esteja fora da função e assim pode gerar uma excessão.

**4 - Interface funcional Consumer**

Exemplo-1: Uso de *Consumer* direto no ***método*** ***forEach()***:

**5 - Interface funcional Function**

public interface Function { R apply(T t); }

- Nota sobre a função map:

- A função "map" *(não confunda com a estrutura de dados Map)* é uma função que aplica uma função a todos elementos de uma stream.

- Conversões: List para stream: .stream()

- Stream para List: .collect(Collectors.toList())

**6 - Stream**

- É uma sequencia de elementos advinda de uma fonte de dados que oferece suporte a "operações agregadas".

**- Fonte de dados:** coleção, array, função de iteração, recurso de E/S.

***- Sugestão de leitura:***[*http://www.oracle.com/technetwork/pt/articles/java/streams-api-java-8-3410098-ptb.html*](http://www.oracle.com/technetwork/pt/articles/java/streams-api-java-8-3410098-ptb.html)

**- Características**

- Stream é uma solução para processar sequências de dados de forma:

**- Declarativa** (iteração interna: escondida do programador) - Você vai escrever o que fazer e o processamento vai acontecer internamente.

**- Parallel-friendly** (imutável -> thread safe)

**- Sem efeitos colaterais**

**- Sob demanda** *(lazy evaluation)*

**- Acesso sequencial** *(não há índices)*

- **Single-use:** só pode ser "usada" uma vez

- **Pipeline:** operações em streams retornam novas streams. Então é possível criar uma cadeia de operações *(fluxo de processamento).*

**- Operações intermediárias e terminais - O pipeline** é composto por zero ou mais operações intermediárias e uma terminal.

**- Operação intermediária:**

- Produz uma nova streams (encadeamento).

- Só executa quando uma operação terminal é invocada (lazy evaluation).

**- Operação terminal:**

- Produz um objeto não-stream (coleção ou outro).

- Determina o fim do processamento da stream

**- Algumas operações intermediárias:**

- filter - map - flatmap - peek - distinct - sorted - skip - limit (\*)

*\* short-circuit - São operações que quando a condição dela é satisfeita o processamento para. No caso de limit significa que você pode definir a qtde de elementos limite dentro dos parêntesis.*

**- Algumas operações terminais:**

- forEach - forEachOrdered - toArray - reduce - collect - min - max - count - anyMatch (\*) - allMatch (\*) -

noneMatch (\*) - findFirst (\*) - findAny (\*)

*\* short-circuit - São operações que quando a condição dela é satisfeita o processamento para.*

**- Como criar uma stream:**

- Basta chamar o *método stream()* ou *parallelStream()* a partir de qualquer *objeto Collection*.

*(Ver:* [*https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html*](https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html)*)*

- Outras formas de se criar uma stream incluem:

***- Stream.of***

*//Criar uma 'Stream' usando o método 'Stream.of()'*

*Stream<String> st2 = Stream.of("Maria", "Alex", "Bob");*

*[Maria, Alex, Bob]*

***- Stream.ofNullable***

***- Stream.iterate***

***//Exemplo-1***

*System.out.println("Criar uma 'Stream<Integer> st3' usando o método 'iterate()'");*

*System.out.println("partindo de '0'(zero) e somando +2 até o limite de 10 elementos");*

*System.out.println("Stream<Integer> st3 = Stream.iterate(0, x -> x + 2);");*

*System.out.println("System.out.println(Arrays.toString(st3.limit(10).toArray()));");*

*Stream<Integer> st3 = Stream.iterate(0, x -> x + 2);*

*System.out.println(Arrays.toString(st3.limit(10).toArray()));*

***//Exemplo-2***

*//Criar uma 'Stream' usando o método 'iterate()' representando*

*//a sequência de Fibonacci.*

*System.out.println("Criar uma 'Stream' usando o método 'iterate()'");*

*System.out.println("representando a sequência de Fibonacci");*

*System.out.println("até o limite de 20 elementos.");*

*System.out.println("Stream<Long> st4 = Stream*

*.iterate(new Long[] {0L, 1L}, p -> new Long[] {p[1], p[0] + p[1]})*

*.map(p -> p[0]);");*

*System.out.println("System.out.println(Arrays.toString(st4.limit(20).toArray()));");*

*Stream<Long> st4 = Stream*

*.iterate(new Long[] {0L, 1L}, p -> new Long[] {p[1], p[0] + p[1]})*

*.map(p -> p[0]);*

*System.out.println(Arrays.toString(st4.limit(20).toArray()));*

**- Copiar os dados de um *List* para uma *stream***

*//Instancia uma variável 'list' tipo 'List<Integer>'.*

*List<Integer> list = Arrays.asList(3, 4, 5, 10, 7);*

*Stream<Integer> st1 = list.stream();*

**- Usando o método *map()***

- O ***map*** aplica uma **função** a cada elemento da string, neste caso multiplica cada elemento por 10.

*Stream<Integer> st1 = list.stream().map(x -> x \* 10);*

*[30, 40, 50, 100, 70]*

**- Usando o método reduce()**

O método ***'reduce()***' aplica uma ***função*** partindo de um ***elemento neutro inicial*** e em seguida pega dois argumentos e aplica uma ***função***.

**- Exemplos:**

*System.out.println("Sintaxe: int sum = list.stream().reduce(0, (x, y) -> x + y);");*

*System.out.println("---------------------------------------------------");*

*int sum = list.stream().reduce(0, (x, y) -> x + y);*

*System.out.println("Soma = " + sum);*

*System.out.println("---------------------------------------------------");*

*System.out.println("Sintaxe: int multipl = list.stream().reduce(1, (x, y) -> x \* y);");*

*int multipl = list.stream().reduce(1, (x, y) -> x \* y);*

*System.out.println("---------------------------------------------------");*

*System.out.println("Multiplicação = " + multipl);*

**- Usando o Pipeline**

- Criar uma **'list2'** usando o **método** **'filter()'** p/ filtrar os números pares e o **método** **'map()'** p/ multiplicar estes números por 10.

*System.out.println("List<Integer> list2 = list.stream()\n"*

*+ ".filter(x -> x % 2 == 0)\n"*

*+ ".map(x -> x \* 10)\n"*

*+ ".collect(Collectors.toList());");*

*//Exibir usando a solução simplificada 'text block'.*

*//System.out.println("""*

*//List<Integer> list2 = list.stream()*

*//.filter(x -> x % 2 == 0)*

*//.map(x -> x \* 10)*

*//.collect(Collectors.toList());""");*

*System.out.println("---------------------------------------------------");*

*List<Integer> list2 = list.stream()*

*.filter(x -> x % 2 == 0)*

*.map(x -> x \* 10)*

*.collect(Collectors.toList());*

*System.out.println("list2 = " + Arrays.toString(list2.toArray()));*

**26) - JDBC - Java Database Connectivity**

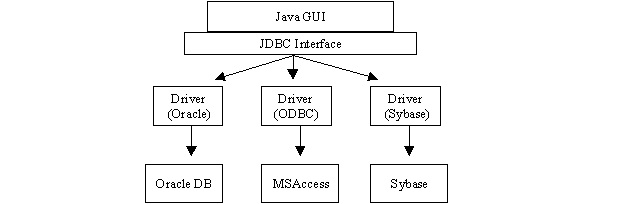
**1) - API padrão do Java para acesso a dados**

**- Páginas oficiais:**

<https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>

<https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html>

Pacotes: java.sql e javax.sql (API suplementar para servidores)



🡸 JDBC Driver Manager

- O **JDBC Driver Manager** converte o código do usuário para os bancos de dados específicos

**2) - Criar uma conexão com banco de dados**

**3) - Recuperar dados**

**4) - Inserir dados**

**5) - Atualizar dados**

**6) - Deletar dados**

**7) - Transação segura**

- Como implementar uma transação usando o JDBC. Transação é uma operação que tem que manter a consistência do banco de dados e são 04 propriedades que ela deve ter:

**- ACID :** *Atomicidade - - Consistência - Isolamento - Durabilidade*

*-* ***Atomicidade* -** Todas as alterações nos dados são realizadas como se fossem uma única operação. Ou seja, todas as mudanças são realizadas, ou nenhuma delas. Por exemplo, em um aplicativo que transfere fundos de uma conta para outra, a propriedade atomicity garante que, se um débito for feito com sucesso a partir de uma conta, o crédito correspondente é feito para a outra conta.

***- Consistência* -** Os dados estão em um estado consistente quando uma transação começa e quando termina.

Por exemplo, em uma aplicação que transfere fundos de uma conta para outra, o imóvel de consistência garante que o valor total dos fundos em ambas as contas seja o mesmo no início e no final de cada transação.

***- Isolamento* -**  O estado intermediário de uma transação é invisível para outras transações. Como resultado, as transações que são executadas simultaneamente parecem ser serializadas. Por exemplo, em um aplicativo que transfere fundos de uma conta para outra, o imóvel de isolamento garante que outra transação veja os fundos transferidos em uma conta ou outra, mas não em ambas, nem em nenhuma delas.

***- Durabilidade* -** Após uma transação concluída com sucesso, as alterações nos dados persistem e não são desfeitas, mesmo em caso de falha no sistema.

Por exemplo, em um aplicativo que transfere fundos de uma conta para outra, o imóvel de durabilidade garante que as alterações feitas em cada conta não serão revertidas.

- Vamos usar basicamente 03 elementos da API do JDBC:

***- setAutoCommit (false)*** - Cada operação isolada que for feita ela vai ser confirmada automaticamente caso entre parêntesis seja ‘true’.

***- commit()*** - Significa confirmar a transação

***- rollback()*** - Significa desfazer o que foi feito até o momento caso aconteça uma falha no meio da transação.

- ***Exemplo de código:*** ***Projeto: Ex21\_259\_JDBC6\_Transacao\_Segura***

***Demonstrando o uso de 'setAutoCommit(false)' / commit() / rollback()***

**8) - Padrão de projeto DAO - Data Access Object**

*- Referências:* [*https://www.devmedia.com.br/dao-pattern-persistencia-de-dados-utilizando-o-padrao-dao/30999*](https://www.devmedia.com.br/dao-pattern-persistencia-de-dados-utilizando-o-padrao-dao/30999)

O padrão de projeto DAO surgiu com a necessidade de separarmos a lógica de negócios da lógica de persistência de dados. Este padrão permite que possamos mudar a forma de persistência sem que isso influencie em nada na lógica de negócio, além de tornar nossas classes mais legíveis.

Classes DAO são responsáveis por trocar informações com o SGBD e fornecer operações CRUD e de pesquisas, elas devem ser capazes de buscar dados no banco e transformar esses em objetos ou lista de objetos, fazendo uso de listas genéricas (**BOX 3**), também deverão receber os objetos, converter em instruções SQL e mandar para o banco de dados.

Toda interação com a base de dados se dará através destas classes, nunca das classes de negócio, muito menos de formulários.

Se aplicarmos este padrão corretamente ele vai abstrair completamente o modo de busca e gravação dos dados, tornando isso transparente para aplicação e facilitando muito na hora de fazermos manutenção na aplicação ou migrarmos de banco de dados.

Também conseguimos centralizar a troca de dados com o SGBD (Sistema Gerenciador de Banco de Dados), teremos um ponto único de acesso a dados, tendo assim nossa aplicação um ótimo design orientado a objeto.

**- Ideia geral do padrão DAO:**

- Para cada entidade, haverá um objeto responsável por fazer acesso a dados relacionado a esta entidade. *- Por exemplo:* *Cliente: ClienteDao*

*Produto: ProdutoDao*

*Pedido: PedidoDao*

- Cada **DAO** será definido por uma ***interface*** para que o sistema fique flexível e preserve o contrato de objetos de acesso a dados, visto que o acesso a dados pode mudar no futuro próximo, por exemplo, hoje pode ser um banco de dados MySQL e amanhã migrar para um ORACLE, ou hoje se usar uma tecnologia de acesso a dados tipo o JBDC *(Java)* ou ADO *(C#)* e amanhã migrar para um ORM - Object- Relational Mapping *(Mapeamento de Objeto Relacional).*

- A injeção de dependência pode ser feita por meio do padrão de ***projeto Factory*** o qual é o responsável pela implementação do DAO.

**100) - Bônus Git e Github**

**Git e Github**

**- Git** - é um sistema de versionamento que instalamos no nosso computador, onde podemos controlar as modificações de um projeto por meio de versões chamadas ‘*commits’*. Cada *commit* tem um código identificador único.

**- Github** - É um site de serviço online de hospedagem de projetos via web gerenciado pelo Git.

- É um serviço online de hospedagem de repositórios Git remotos.

- Possui uma interface gráfica web: github.com

- É uma plataforma social *(usuários, página de perfil, seguidores, colaboração, etc.).* Dica: currículo!

- Maior serviço do mundo de hospedagem de projetos de código aberto

- Modelo de cobrança: gratuito para projetos de código aberto, pago para projetos privados

- Alternativas: BitBucket, GitLab, etc.

**- Repositório remoto e local** - Um projeto controlado pelo Git é chamado de repositório de versionamento.

**- Repositório remoto** - Tipicamente uma cópia "oficial" do repositório fica salvo em um servidor que é o repositório remoto.

**- Repositório local** - Cada pessoa que trabalha no projeto pode fazer uma cópia do repositório para seu computador *(repositório local).* A pessoa então faz suas alterações no projeto *(novos commits)* e depois salva as alterações no servidor.

***- Fluxo***

- 1) - Fazer uma cópia do *repositório remoto* para o *repositório local*, na prática, é usar o comando *git.pull* ou *git.clone*. Normalmente a primeira cópia que se faz é um *git.clone* pois não ainda não existe uma cópia no computador local.

- 2) - Durante o dia de trabalho digamos que o programador salvou 02 commits (versões) no repositório local, deste modo no final do dia, ele usa o comando *git.push* para salvar estas 02 versões no repositório remoto do servidor.

**1) - Configurar chave SSH para o Github**

***- SSH*** - É um protocolo para comunicação de dados de segurança

- O Github aboliu a autenticação somente com usuário e senha.

- A ideia básica é cadastrar previamente quais computadores podem acessar o Github em seu nome.

- Outros computadores não conseguem acessar.

**- Para isso é preciso:**

- 1) - Gerar uma chave SSH no seu computador

- 2) - Cadastrar essa chave no seu Github

**2) - Comandos no terminal Git Bash**

1 **- $ clear -** Limpa a tela de comandos Git Bash

2 **- $** **git config --list** - Exibe o nome de usuário e email cadastrados.

3 **- $ ls** - Exibe os arquivos de uma pasta

4 **- $ git log** - Exibe os detalhes das versões salvas

5 **- $ git log --oneline** - Exibe apenas uma linha com descrição da versão salva

6 **- $ git status** - Pode exibir ***03 situações***:

***- modified*** - Exibe os arquivos que foram modificados

***- untracked*** - Exibe eventual arquivo novo criado *(não rastreado),* ou seja, ainda não foi commitado.

***- deleted*** - Exibe eventual arquivo deletado

7 **- $ git add .** - ***Adiciona*** ***todos os arquivos*** *(note o ponto (.) após a palavra add),*  *novos, modificados ou deletados* na área de *stage,* que é uma área onde são armazenados os arquivos antes do comando $ commit, e neste caso cria uma única versão para todos os arquivos que estavam na área de stage.

8 **- $ git add <arquivo a ser commitado>** - Adiciona na área de stage apenas o arquivo a ser commitado, ou seja, pode-se fazer várias alterações e salvar versões parciais.

9 **- $ git reset** - Retira os arquivos incluídos na área de stage pelo comando **$ add .** e volta a situação dos mesmos para antes do comando.

10 - **$ git commit -m “***<Descrição da versão salva>***”** - Salva no *github.com* a(s) versão(ões) do(s) arquivo(s) que estavam na área de *stage*.

11 - **$ git diff** - Exibe textualmente as modificações feitas em um arquivo.

12 - **$ git checkout** *<codigo>* - Permite modificar temporariamente para o estado *head* uma versão dos arquivos de um commit ou branch que for identificado pelo *<código>*.

13 - **$ git checkout** <head~1> - Permite modificar temporariamente para o estado *head* uma versão dos arquivos de um commit ou branch que for identificado por *<head~1 ou head~2, etc.>*. Neste casi não é necessário especificar o código basta a palavra *head~n.*

14 - **$ git clean -df** - **Passo-1** para desfazer modificações não salvas. Deve ser usado antes de um *$ git checkout <head> ou <código)*. Limpa as eventuais modificações feitas no arquivo que está no *head*.

15 - **$ git checkout -- .** - **Passo-2** para desfazer modificações não salvas. Análogo ao *$ git clean -df*. Deve ser usado antes de um *$ git checkout <head> ou <código)*. Limpa as eventuais modificações feitas no arquivo que está no *head*.

16 - **$ git reset --soft <***HEAD~1>* - Deleta o último commit sem apagar as modificações.

17 - **$ git reset --hard** *<código commit>* - Deleta os commits subsequentes a esse código.

18 - **$ git pull origin main** - Carrega no *repositório local* tudo que está armazenado no *repositório remoto*.

- Nota: *<origin>* nome do repositório remoto e *<main>* nome do *bancch*.

**3) - Passo a passo: salvar primeira versão de um projeto no Github**

Considerando que agora seu ambiente já está todo configurado *(usuário e email, visualização de arquivos ocultos, chave SSH)*, sempre que você criar um novo projeto, os passos básicos serão estes (troque os parâmetros em azul pelos seus dados):

Passo-1) - Criar uma pasta no *explorador de arquivos* do Windows para armazernar o projeto, ex.: *site-de- vendas.*

Passo-2) - Abrir o *Visual Code* e inserir esta pasta *site-de-vendas* lá.

Passo-3) - No Visual Code, dentro da pasta *site-de-vendas*, criar 02 arquivos html: *índex.html e vendas.html.*

- ***Nota:*** No Visual code ***digite <!>*** e ***Enter*** para criar uma estrutura básica de código HTML.

Passo-4) - ***$ git init*** - Cria um repositório Git na pasta *site-de-vendas* no *explorador de arquivos*, ou seja, esta pasta vai ser um projeto gerenciado pelo Git. Note que foi criado uma subpasta *.git* que será gerenciada pelo próprio Git, portanto não se deve tocar nesta subpasta.

Passo-5) - ***$ git add*** . - Envia os arquivos para uma área temporária chamada ‘*stage’* antes de salvar no repositório local.

Passo-6) - ***$ git status*** - Exibe os arquivos que estão na área ‘stage’ prontos para serem salvos

Passo-7) - ***$ git commit -m*** ***"Criadas as páginas índex e vendas"*** - Salva no repositório local Git a nova versão a ser enviada para o repositório remoto.

Passo-8) ***- $ git branch*** ***-M main*** - Atualmente o Github usa por padrão o nome *main* para salvar o histórico de commits. Antes era usado o nome *master.*

Passo-9) - ***Abrir o site do Github:*** <https://github.com/>, clicar no *ícone do gatinho* para ir para a *tela principal* e em seguida clicar no botão ***New*** para criar um novo repositório no Github***.***

Passo-10) - ***No campos à frente do campo do usuário*** digitar um nome para o repositório: *aulagithub*. Não marcar mais nada na tela para que seja criado um projeto vazio. Clicar no botão ***Criar Repositório.***

Passo-11) ***- $ git*** ***remote add origin*** [git@github.com:Antonio-1955/aulagithub.git](mailto:git@github.com:Antonio-1955/aulagithub.git) ***-***  Faz a associação entre o projeto que está no repositório local do computador com o projeto do repositório remoto do Github. A parte em azul *(link)* foi gerada no site do Github quando da criação do projeto nos passos 9 e 10. Note que é presiso clicar no botão *SSH* e não *HTTPS*.

Passo-12) ***- $ git push -u origin main*** - Envia o projeto do repositório local para o repositório remoto do Github. Note que como este é o primeiro push com a chave SSH criada, o sistema pergunta: *Are you sure you want do continue connecting (Yes/no/[fingerprint])?*

Passo-13) - ***Processo finalizado: 1 commit salvo***.

**4) - Passo a passo: salvar uma nova versão do projeto no Github**

- Passo 1) - ***Modificar o projeto site-de-vendas***acrescentando um parágrafo *<p>Vendas = R$ 5000,00</p>* no arquivo *vendas.html*

- Passo 2) - ***$ git status***

*- on branch main*

*Your branch is up to date with 'origin/main'.*

*Changes to be committed:*

*(use "git restore --staged <file>..." to unstage)*

*modified: vendas.html*

- Passo 3) - ***$ git add .*** - Envia o arquivo modificado para uma área temporária chamada ‘*stage’* antes de salvar no *repositório local*.

- Passo 4) - ***$ git commit -m “Acrescentado valor das vendas”*** - Salva no *repositório local* *Git* a nova versão a ser enviada para o *repositório remoto*.

*- [main 5dfb095] Acrescentado valor das vendas*

*1 file changed, 1 insertion(+)*

- Passo 5) - ***$ git push*** - Envia o projeto do *repositório local* para o *repositório remoto* do *Github*.

*- Enumerating objects: 5, done.*

*Counting objects: 100% (5/5), done.*

*Delta compression using up to 8 threads*

*Compressing objects: 100% (3/3), done.*

*Writing objects: 100% (3/3), 347 bytes | 347.00 KiB/s, done.*

*Total 3 (delta 1), reused 0 (delta 0), pack-reused 0*

*remote: Resolving deltas: 100% (1/1), completed with 1 local object.*

*To github.com:Antonio-1955/aulagithub.git*

*937dc59..5dfb095 main -> main*

- Passo 6) - ***Processo de modificação finalizado***: 02 commits salvos.

**5) - Passo a passo: clonar/modificar projeto de um repositório remoto que você tem permissão para alterar**

- Passo 1) - Entrar no <https://github.com/Antonio-1955/aulagithub>.

- Passo 2) - Clicar no botão *Code*, escolher *SSH* e copiar o endereço <git@github.com:Antonio-1955/aulagithub.git>

- Passo 3) - Abrir no Explorador de arquivos do Windows a pasta onde o projeto será clonado.

- Passo 4) - Dentro da pasta clicar com o botão direito e escolher a opão *Git Bash* p/ abrir a janela de comandos.

- Passo 5) - Na janela de comandos digitar: *$ git clone* e colar o endereço copiado no passo 2)

[git@github.com:Antonio-1955/aulagithub.git](git@github.com:Antonio-1955/aulagithub.git%20) + Enter.

- Passo 6) - ***$ cd aulagithub/*** - entrar no subdiretório do arquivo clonado.

- Passo 7) - ***$ code .*** - para abrir o *Visual Code* diretamente da tela de comando.

- ***Nota:*** No Visual code ***digite <!>*** e ***Enter*** para criar uma estrutura básica de código HTML.

- Passo 8) - Modificar o projeto com o *Visual Code* criando o arquivo *blog.html* .

- Passo 9) - ***$ git status*** - Exibe o arquivo que foi criado

*On branch main*

*Your branch is up to date with 'origin/main'.*

*Untracked files:*

*(use "git add <file>..." to include in what will be committed)*

***blog.html***

*nothing added to commit but untracked files present (use "git add" to track)*

Passo 10) - ***$ git add .*** para colocar o arquivo na área stage

Passo 11) - ***$ git commit -m*** para salvar a nova versão no repositório local.

Passo 12) - ***$ git log*** para se visualizar as versões do repositório local.

Passo 13) - Repetir os passos 8, 10, 11 e 12 para salvar mais uma versão e mostrar o log.

Passo 14) - ***$ git push*** - Envia o projeto, contendo as duas mofificaçoes, do *repositório local* para o *repositório remoto* do *Github.*

Passo 15) - ***$ git log --oneline*** - Exibe os logs, um por linha, contendo apenas as descrições das modificaçoes.

Passo 16) - ***Processo de clonar/modificar finalizado***: 02 commits salvos.

**6) - Arquivo .gitignore**

- É um arquivo que indica o que NÃO deve ser salvo pelo Git.

- Geralmente o arquivo .gitignore fica salvo na pasta principal do repositório. Mas também é possível salvar outros arquivos .gitignore em subpastas do repositório, para indicar o que deve ser ignorado por cada subpasta.

***- Casos comuns de arquivos que não devem ser salvos pelo Git:***

*- Arquivos compilados* - Linguagens compiladas (C, C++, Java, C#, etc.) geram arquivos de código compilado para executar o programa localmente.

*- Arquivos de bibliotecas externas usadas no projeto* - Projetos reais utilizam bibliotecas externas (que são programas prontos disponíveis na Internet). Por exemplo, projetos JavaScript com NPM tipicamente salvam uma subpasta "node\_modules" na pasta do seu projeto.

*- Arquivos de configuração da sua IDE* - IDE’s podem salvar uma subpasta com arquivos de configuração na pasta do projeto (esemplo: .vscode).

*- Arquivos de configuração do seu sistema* - Por esemplo, sistemas Mac podem gravar uma subpasta .ds\_store na pasta do projeto.

***- Exemplos de projetos com .gitignore***

- <https://github.com/acenelio/composition1-java>

- <https://github.com/acenelio/dsmovie>

**7) - O que fazer quando abre o editor VIM - É o terminal editor básico do Linux**.

*-* ***Estas ações pdem abrir o editor VIM no terminal:***

- Fazer um commit sem mensagem.

- Fazer um merge de três vias.

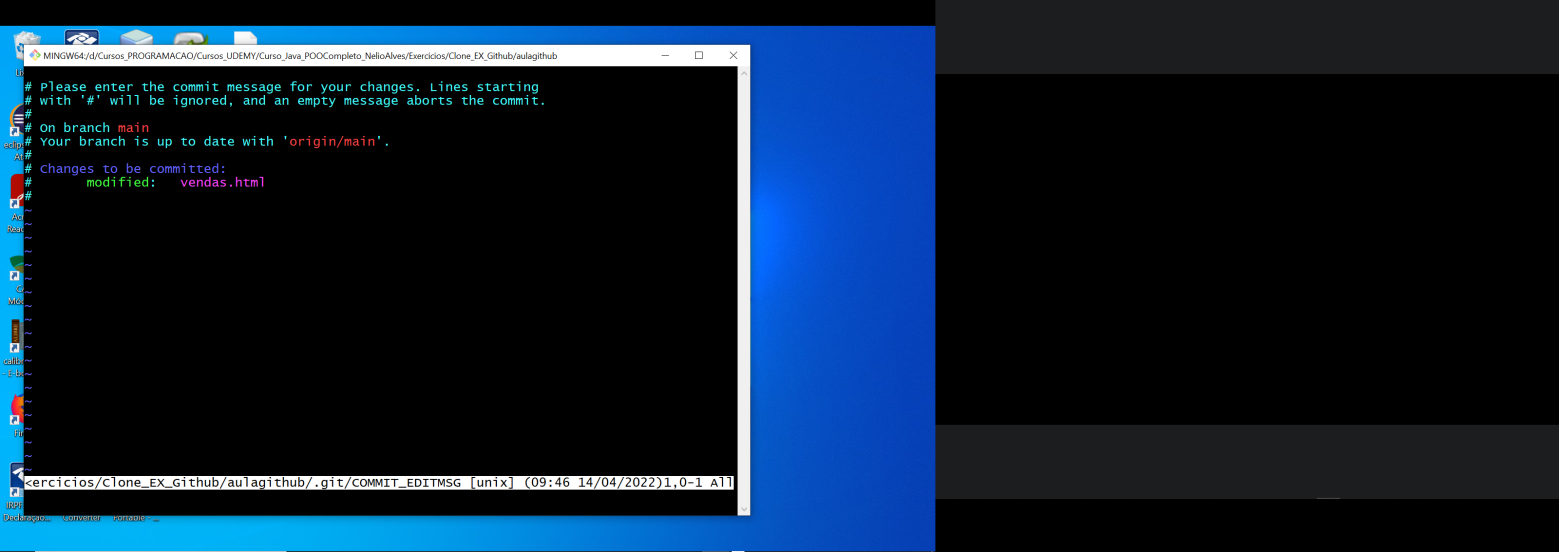
- Exemplo de fazer um commit sem mensagem:

- Modificar o arquivo vendas.html deletando uma linha.

- Digitar no Git bash o comando $ git add .

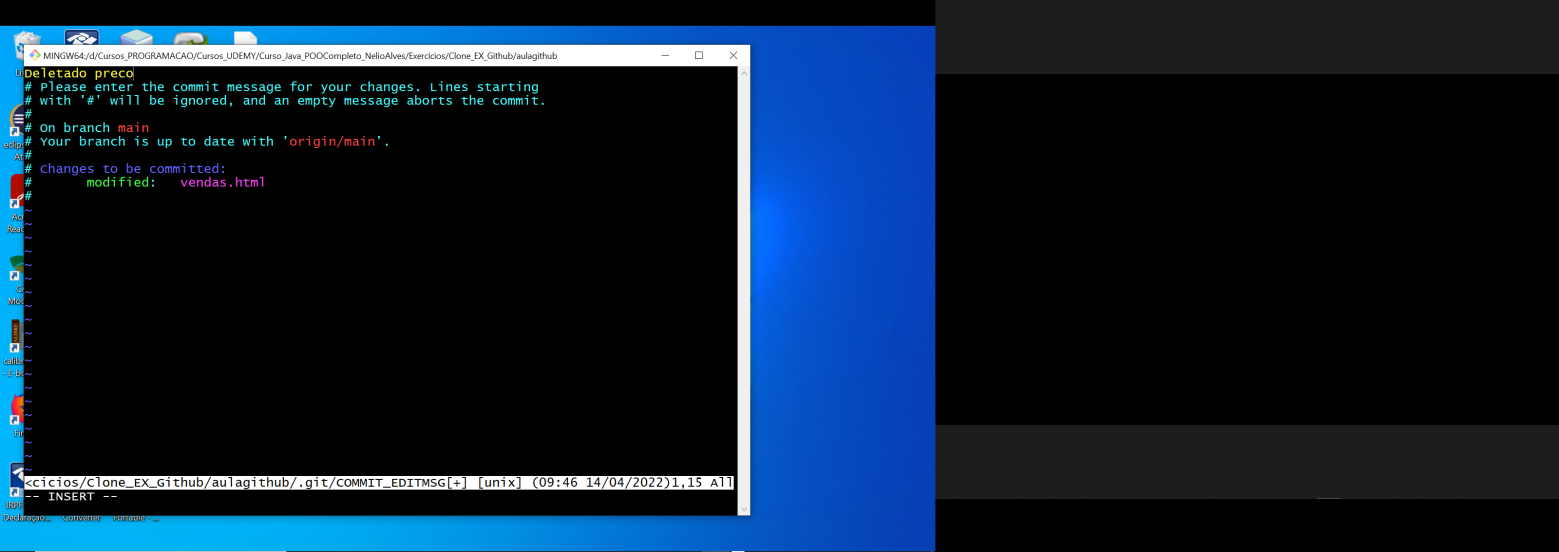
- Em seguida digitar o comando $ git commit sem especificar uma mensagem que identifica a modificação.

- Imediatament aparece a tela do editor VIM abaixo:



- O editor VIM pede para entrar com a mensagem e para isso é preciso abrir o editor pressionando a tecla <i>.

- Logo abaixo aparece -- INSERT -- para que possa ser digitada a msg *Deletado preco*



- Para sair do VIM salvando teclar <ESC> e digitar <:wq> *(:Wright quit)* e teclar <ENTER>

- Para sair do VIM sem salvar teclar <ESC> e digitar <:q!> *(:quit !)* e teclar <ENTER>

- O commit é salvo e volta para a tela do Git Bsh.

**8) - Como resolver push rejeitado por históricos diferente**

- Não é permitido enviar um *push* se seu repositório local está atrasado em relação ao histórico do repositório remoto Por exemplo:

***- No Github tem 03 versões oficiais (commits)***

V2

V3

V1

***- No repositório local tem 02 versões (commits) oficiais + 02 que não existem no repositório remoto (Github).***

x4

x3

V2

V1

- Ao se tentar dar um comando *$ git push* para atualizar o *repositório remoto* será mostratda a mensagem de *push rejeitado* devido à incongruência entre os repositórios.

*- $ git push*

*-To github.com:Antonio-1955/site\_biblioteca\_teste.git*

*! [rejected] main -> main (fetch first)*

***error: failed to push some refs to 'github.com:Antonio-1955/site\_biblioteca\_teste.git'***

*hint: Updates were rejected because the remote contains work that you do*

*hint: not have locally. This is usually caused by another repository pushing*

*hint: to the same ref. You may want to first integrate the remote changes*

*hint: (e.g., 'git pull ...') before pushing again.*

*hint: See the 'Note about fast-forwards' in 'git push --help' for details.*

- Ou seja, é preciso respeitar o histórico, portanto para evitar o *push rejeitado* é preciso fazer um ***$ git pull origin main***, trazendo para o *repositório local* a versão do *remoto*.

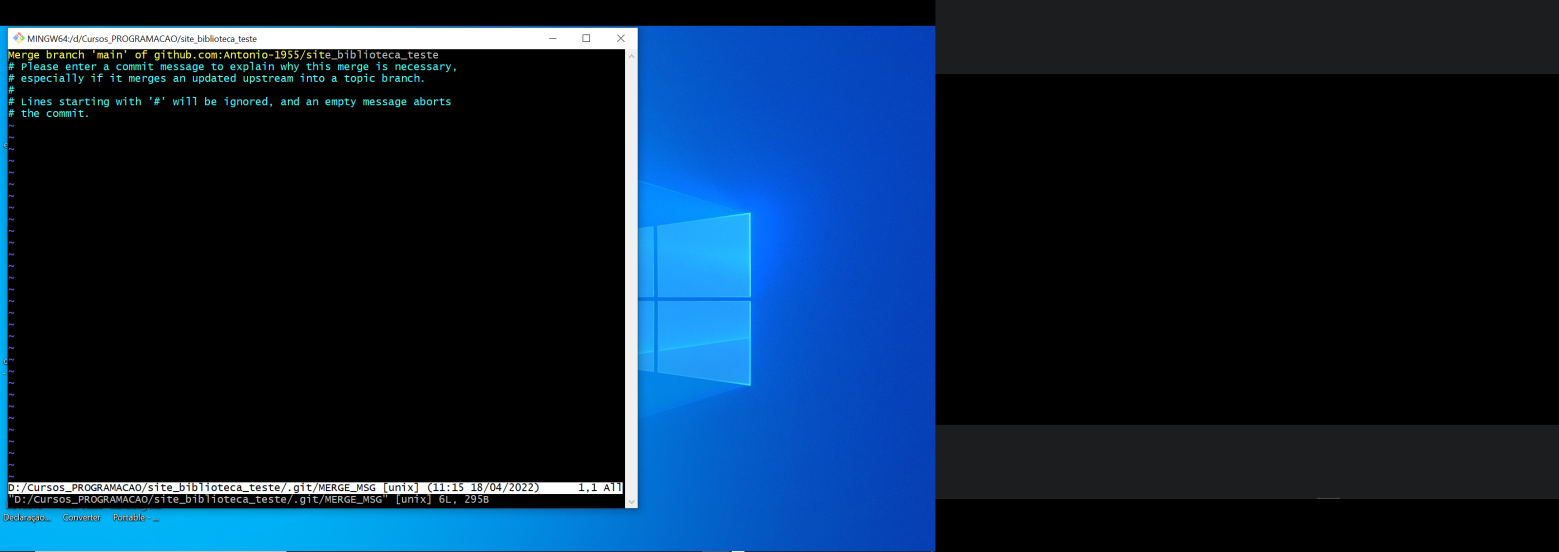
- Depois do comando é aberta a página de *VIM* *(ver o primeiro print),* porque está sendo trazido um *commit* que está no meio do caminho *(chamado de merge de 03 vias),* é preciso fazer um *commit* na frente para consolidar esta modificação.

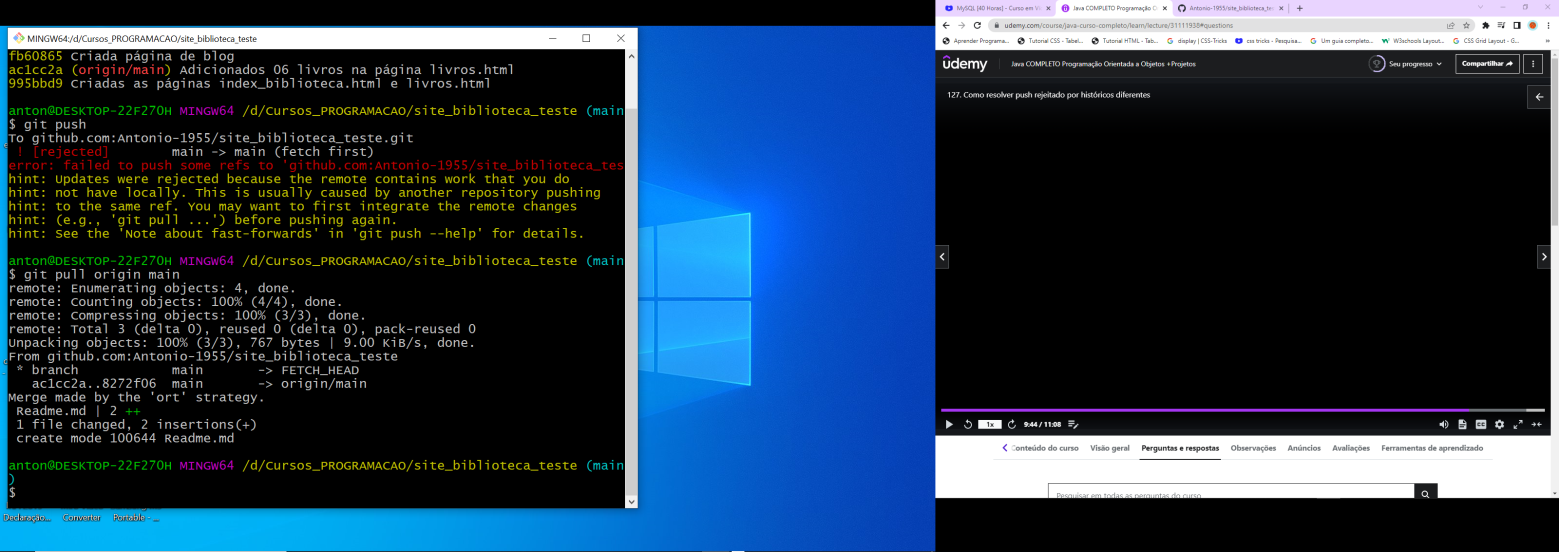
- Para isso basta entrar no modo de edição digitando <i> - e se quiser modificar a linha amarela basta adicionar uma mensagem própria ou deixar a que está indicada. Em seguida apertar a tecla *<ESC>,* digitar *dois pontos wq* <:wq> *(write quit)* para salvar e sair + *<ENTER>.*

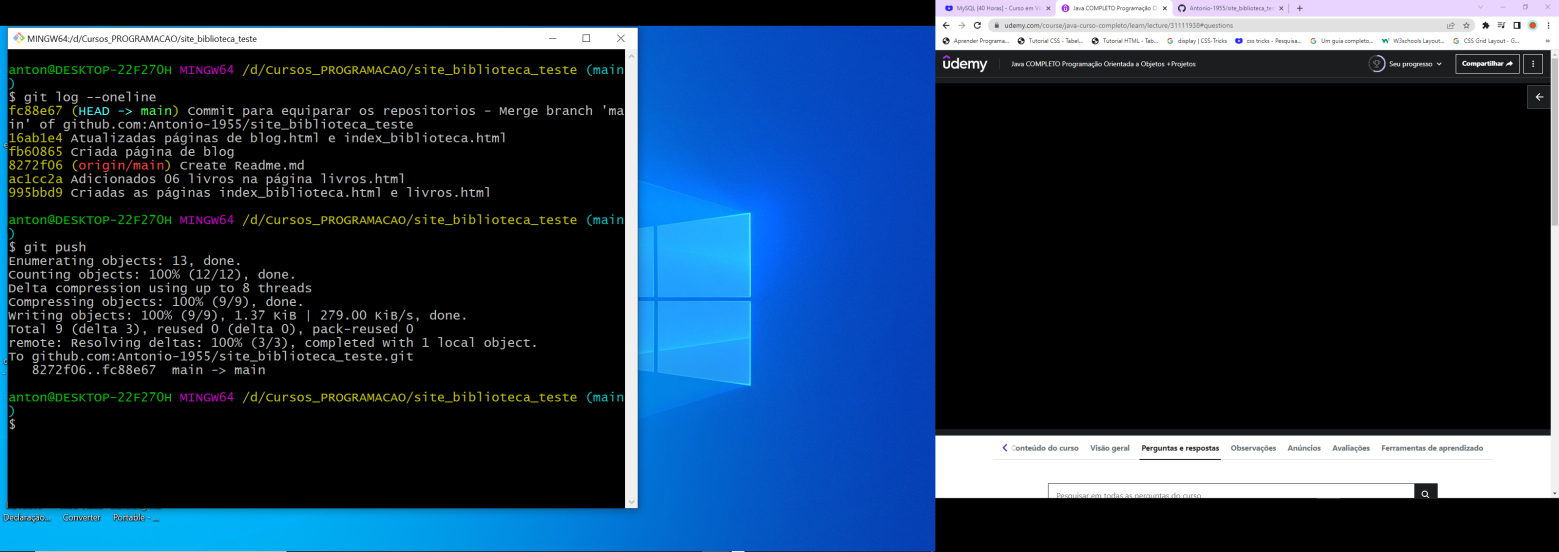
- Vai ser executado o comando e mostrar o *segundo print abaixo.*

- Ai é só fazer o *$ git push*, que antes havia sido rejeitado, para atualizar o repositório remoto.

- Ver o resultado no terceiro print abaixo.







**9) - Resolvendo pull com conflito**

- Um conflito acontece quando se vai mesclar *(merging)* 02 históricos e estes históricos diferentes mexeram no mesmo arquivo. Neste caso o Git não sabe qual é o correto e dá as indicações para que o conflito seja resolvido.

- Isto acontece quando você atualiza um arquivo no repositório local e ele possui uma outra modificação no *remoto* que ainda não foi trazida para o *local*, ou seja o *local* está desatualizado em relação ao *remoto*.

- Abaixo está descrito como a sequência de como o conflito se dá:

- 1 - O arquivo *log.html* é modificado no local adicionando o *Blog-4).*

- 2 - No *Git* você dá o comando $ git status e ele mostra que o este arquivo está em *“changes not staged for commit”*.

- 3 - Você dá o comando *$ git add .* e em seguida o *$ git commit -m “Adicionado Blog-4)”*

- 4 - Em seguida você dá o *$ git push* e recebe a mensagem de erro ***<! [rejected] error: failed to push some refs to ‘...’*** e as dicas para dar um comando *$ git pull* e trazer a versão atualizada o *repositório remoto* para o *local*.

- 5 - Quando você dá o *$ git pull origin main* para buscar no *repositório remoto do Github* a atualização que não está no *local* do seu *computador*, o *Git* mostra que você está no meio de um *<main|MERGING>* e entre outras informações mostra a mensagem:

*CONFLICT (content): Merge conflict in bloc.html - Automatic merge failed; fix conflicts and then commit the result.*

- 6 - Dando um comando *$ git status* será mostrado que existe uma divergência e que é preciso resolver o conflito.

- 7 - Abrindo o *código-fonte* de *blog.html* no *VS code* será exibido as linhas em conflito, ai é só escolher a correta e apagar o que está incorreto e salvar.

- 8 - Ai é só dar um *$ git add .,* depois um *$ git commit -m “...”* e em seguida um *$ git push* e o conflito está resolvido.

**10) - Como sobrescrever um histórico no Github**

**- Atenção: esta é uma ação DESTRUTIVA**.

- Para exemplificar vamos dar um comando *$ git reset --hard fb60865* para apagar do repositório local todas as versões anteriores à *fb60865*

- Dando o comando *$ git log --oneline* será mostrado que ficaram apenas as 03 primeiras versões incluindo a *fb60865*.

- Neste ponto com o comando *$ git push -f (f significa forced update),* vamos sobrescrever os 09 commits *(versões)* presente no *Github* com os 03 *commits* presentes no *repositório local* do *computador*. Uffa!!

- Pronto, no Github estão apenas os 03 commits.

**11) - Como apontar o projeto para outro repositório remoto no Github**

- Primeiro é preciso criar um novo repositório no Github:

- 1 - Clicar no ícone do *gatinho* e depois em *<New>*

- 2 - Digitar o nome do novo repositório <Testebiblioteca>

- 3 - Copiar o comando *<git remote add origin git@github.com:Antonio-1955/Testebiblioteca.git>*

- 4 - Digitar no *Git* do computador o comando:

*- $ git remote set-url git remote add origin git@github.com:Antonio-1955/Testebiblioteca.git*

- 5 - Pronto, o projeto local está apontando para o novo repositório no Github que agora tem 02 repositórios remotos com as mesmas versões.

**100) - Tópicos a pesquisar** *(citados em livros mas não apresentados)*

1 - Livro USE A CABEÇA JAVA *(Head First Java)* - As únicas maneiras de se criar um novo objeto são

através de “new” ou da desserialização (ou algo chamado de Java Reflection API)

***Resumo Livro Java Como Programar (10ª Edição)***

***Autores: Paul Deitel e Harvey Deitel***

***Aluno autor deste resumo: Antônio G. Campos***

**Capítulo-2 - Introdução a Aplicativos Java - Entrada / Saída e Operadores**

- A ***barra invertida \*** é um caractere de ***escape*** e tem significado no ***método print***

**\n** ***Nova linha***. Posiciona o cursor de tela no início da próxima linha.

**\t** ***Tabulação horizontal***. Move o cursor de tela para a próxima parada de tabulação.

**\r** - ***Retorno de carro***. Posiciona o cursor da tela no início da linha atual — não avança para a próxima linha. Qualquer saída de caracteres depois do retorno de carro sobrescreve a saída de caracteres anteriormente gerada na linha atual.

**\\** ***Barras invertidas***. Utilizadas para imprimir um caractere de barra invertida.

**\"** ***Aspas duplas***. Utilizadas para imprimir um caractere de aspas duplas. Por exemplo, System.out.println("\"entre aspas\""); exibe "entre aspas".

**2.5.1) - Declarações import** *(item do livro)*

- ***pacotes*** — chamados de grupos de classes relacionadas e, coletivamente, são chamadas de biblioteca de classes Java, ou ***Java API - Java Application Programming Interface*** .

- ***import java.util.Scanner;***

- ***Scanner input = new Scanner(System.in);***  🡺 é uma instrução de declaração de variável que especifica o nome (***input***) e o tipo (***Scanner***) de uma variável utilizada nesse programa.

- Um Scanner permite a um programa ler os dados *(por exemplo, números e strings)* para utilização nele.

- Os dados podem ser provenientes de várias origens, como os digitados pelo usuário ou um arquivo do disco. Antes de utilizar um Scanner, você deve criá-lo e especificar a origem dos dados.

- A palavra reservada ***new*** cria um novo objeto do tipo ***Scanner***

**Capítulo-3 - Introdução a classes, objetos, métodos e strings**

- No ***Capítulo 2***, você trabalhou com ***classes***, ***objetos*** e ***métodos*** existentes*.*

***- System.out*** 🡪 É um ***objeto de saída*** padrão predefinido.

***- print, println e printf*** 🡪 São métodos invocados *(chamados)* pelo objeto ***System.out*** para exibir informações na tela.

- Neste capítulo, você aprenderá a ***criar suas próprias classes e métodos***.

- Cada nova ***classe***que você desenvolve torna-se um ***novo tipo***que pode ser ***utilizado*** para ***declarar variáveis*** e delinear ***objetos***.

- Você pode declarar novos tipos de classe conforme necessário; essa é uma razão pela qual o Java é conhecido como uma ***linguagem extensível.***

- A palavra-chave ***public*** *(explicada no Capítulo 8 em detalhes)* é um ***modificador de acesso***.

- Um ***objeto*** tem ***atributos***, implementados como ***variáveis*** de ***instância*** que o acompanham ao longo

da sua vida.

- As ***variáveis*** de ***instância*** ***existem*** ***antes que os métodos sejam chamados por um objeto***, ***enquanto eles são executados e depois que a execução deles foi concluída.***

- Cada ***objeto*** (***instância***) da ***classe*** tem sua ***própria cópia*** das ***variáveis*** de ***instância da classe***.

- Uma ***classe*** normalmente contém um ou mais ***métodos*** que manipulam as ***variáveis*** de ***instância*** pertencentes aos ***objetos*** particulares dela.

- ***Variáveis de instância*** são declaradas ***dentro***de uma declaração de ***classe***, mas ***fora do corpo*** dos ***métodos*** da ***classe***.

**3.1) Modificadores de acesso *public* e *private***

- A maioria das ***declarações*** de ***variável*** de ***instância*** é precedida pela palavra-chave ***private***.

- Da mesma forma que ***public***, ***private*** é um ***modificador de acesso****.*

*-* As ***variáveis*** ou ***métodos*** declarados com o modificador de acesso ***private*** só são acessíveis a ***métodos*** da ***classe*** em que isso ocorre. Assim, a ***variável*** ***name*** só pode ser empregada nos ***métodos*** de cada ***objeto*** ***Account*** (nesse caso, ***setName*** e ***getName***).

- Você verá mais adiante que isso apresenta oportunidades poderosas de ***engenharia de software***.

**3.2) Anatomia do Método *setName* da classe *Account***

***public void setName(String name) {***

***this.name = name;*** *// armazena o nome*

***}***

- ***public*** 🡪 Modificador de acesso o qual pode ser ***public***, ***private*** ou ***protected***

- ***void*** 🡪 Tipo de retorno que neste caso é vazio

- ***setName*** 🡪 Nome do método. Pode ser qualquer nome. Por convenção usa-se ***set*** para ***atribuir*** *(configurar)* e ***get*** para ***obter***.

- ***(String name)*** 🡪 Este é o ***tipo*** e o ***nome*** do ***parâmetro*** recebido como ***argumento*** da chamada do ***método*** pelo ***objeto myAccount*** *(myAccount.setName(theName);)* , mas pode ser qualquer nome, por convensão usa-se o mesmo nome da ***variável de instãncia*** para não haver proliferação de nomes.

- ***this.name*** 🡪 O ***this*** é uma ***palavra-chave*** para ***referenciar*** *(substuir)* o ***objeto*** *(myAccount)* que fez a chamada do ***método***.

- ***= name***  🡪 É o ***parâmetro*** recebido como ***argumento*** e que será atribuido à ***variável*** ***de instância*** *(também chamada de atributo)* ***name.***

***{ }***  🡪 As ***chaves*** são obrigatórias para ***abrir*** e ***fechar*** o ***bloco*** de ***códigos do método***.

**3.3) - A Classe principal *public class LivroJCP\_Cap3\_Ex06\_ContaBancariat***

- Uma ***classe*** que contém um ***método main,*** inicia a ***execução*** de um ***aplicativo Java***.

- Essa ***classe*** é às vezes denominada ***classe driver*** (*ou* **“*classe condutora”****),* pois a execução do programa começa à partir dela chamando outras eventuais classes.

- A ***classe Account*** *não pode* executar por si só, porque ***não contém*** um ***método main*** — se digitar java Account na janela de comando, você obterá um erro indicando ***“Main method not found in class Account”***. - Para corrigir esse problema, você deve ***declarar*** uma ***classe*** ***separada***que contenha um ***método main*** ou ***colocar*** um ***método main*** na ***classe Account***.

- ***null 🡪*** É o***valor inicial******padrão***para ***variáveis String*** ***de instância,*** diferentemente das ***variáveis locais***, que não sãoinicializadas de forma automática. Toda variável de ***instância*** tem umvalor ***inicial padrão*** fornecido pelo ***Java*** quando você não oespecifica. Portanto, nãoé exigido que as ***variáveis de instância***sejam explicitamente inicializadas antes de serem utilizadas em um programa.*.*

- Uma parte ***essencial*** para permitir à ***JVM*** localizar e chamar o ***método main*** a fim de ***iniciar*** a execução do aplicativo é a palavra-chave ***static***, que indica que ***main*** é um ***método static***.

**- O Programa ContaBancaria**

*package livrojcp\_cap3\_ex06\_contabancaria;*

*import java.util.Scanner;*

*public class LivroJCP\_Cap3\_Ex06\_ContaBancaria {*

*public static void main(String[] args) {*

*// cria um objeto Scanner para obter inputTeclado a partir da janela de comando*

*Scanner inputTeclado = new Scanner(System.in);*

*/\*Se o Scanner for declado conforme abaixo indicado entre colchetes, não é necessário fazer o import:*

***[java.util.Scanner inputTeclado = new java.util.Scanner(System.in);]*** *\*/*

*// cria um objeto Account e o atribui a myAccount*

*Account myAccount = new Account();*

*Account myAccount2 = new Account();*

*// exibe o valor inicial do nome (null)* - *null é o valor inicial padrão para variáveis String*

*System.out.printf("Initial name is: %s%n%n", myAccount.getName());*

*System.out.printf("Initial name is: %s%n%n", myAccount2.getName());*

*// solicita e lê o nome*

*System.out.print("Por favor digite o nome: ");*

*String theName = inputTeclado.nextLine(); // lê uma linha de texto*

*System.out.print("Por favor digite o nome2: ");*

*String theName2 = inputTeclado.nextLine(); // lê uma linha de texto*

*myAccount.setName(theName); // insere theName em myAccount*

*myAccount2.setName(theName2); // insere theName em myAccount2*

*System.out.println(); // gera saída de uma linha em branco*

*// exibe o nome armazenado no objeto myAccount*

*System.out.printf("Nome no objeto myAccount é: %s%n", myAccount.getName());*

*System.out.printf("Nome no objeto myAccount2 é: %s%n", myAccount2.getName());*

*}*

*} // fim da classe AccountTest*

**3.4 -Tipos primitivos versus tipos por referência**

- Os ***tipos*** do ***Java*** são divididos em ***primitivos*** e por ***referência***.

- No Capítulo 2, você trabalhou com variáveis do ***tipo int*** — um dos primitivos. Os outros tipos primitivos são ***boolean***, ***byte***, ***char***, ***short***, ***long***, ***float*** e ***double***, cada um dos quais discutiremos neste livro, eles estão resumidos no Apêndice D.

- Todos os tipos ***não primitivos*** são ***por referência****,* assim, as ***classes*** que especificam os ***objetos*** são por ***referência***.

**- O Programa para Comparar e Coerção Temporária de tipos primitivos**

*package livrojcp\_cap3\_ex08\_comparacao;*

*// Figura 2.15: Comparison.java*

*// Compara inteiros utilizando instruções if, operadores relacionais*

*// e operadores de igualdade.*

*import java.util.Scanner; // programa utiliza a classe Scanner*

*public class LivroJCP\_Cap3\_Ex08\_Comparacao {*

*public static void main(String[] args) {*

*// cria Scanner para obter entrada a partir da linha de comando*

*Scanner inputTeclado = new java.util.Scanner(System.in);*

*//Pode-se usar a declaração abaixo sem fazer o import abaixo de package.*

*//java.util.Scanner inputTeclado = new java.util.Scanner(System.in);*

*int number1; // primeiro número a comparar*

*int number2; // segundo número a comparar*

*System.out.print("Enter first integer: "); // prompt*

*number1 = inputTeclado.nextInt(); // lê o primeiro número fornecido pelo usuário*

*System.out.print("Enter second integer: "); // prompt*

*number2 = inputTeclado.nextInt(); // lê o segundo número fornecido pelo usuário*

*System.out.println("-------------------------------------------------");*

*System.out.println("Voce digitou os números: " + number1 + " e " + number2);*

*System.out.println("-------------------------------------------------");*

*if (number1 == number2) {*

*System.out.printf("%d == %d%n", number1, number2);*

*}*

*if (number1 != number2) {*

*System.out.printf("%d != %d%n", number1, number2);*

*}*

*if (number1 < number2) {*

*System.out.printf("%d < %d%n", number1, number2);*

*}*

*if (number1 > number2) {*

*System.out.printf("%d > %d%n", number1, number2);*

*}*

*if (number1 <= number2) {*

*System.out.printf("%d <= %d%n", number1, number2);*

*}*

*if (number1 >= number2) {*

*System.out.printf("%d >= %d%n", number1, number2);*

*}*

*System.out.println("-------------------------------------------------");*

*//Conversão explícita ou coerção TEMPORÁRIA de tipos primitivos.*

*int a = number1;*

*int b = number2;*

*int x = number1;*

*int y = number2;*

*// Abaixo é aplicado o OPERADOR UNÁRIO DE COERÇÃO temporária de 'int a' para '(double) a'.*

*double media = (double) a / b;*

*System.out.println("Média COM coerção = " + media);*

*// Abaixo NÃO é aplicado o OPERADOR UNÁRIO DE COERÇÃO temporária de 'int a' para '(double)a'.*

*double media2 = (x / y);*

*System.out.println("Média SEM coerção = " + media2);*

*}//Fim Método main*

*}//Fim da Classe*

**3.5 - Construtores não podem retornar valores**

- Uma ***diferença importante*** entre ***construtores*** e ***métodos*** é que os construtores ***não*** ***podem*** ***retornar*** ***valores***, portanto, ***não podem*** ***especificar*** um tipo de ***retorno*** (nem mesmo ***void***).

-Normalmente, os ***construtores*** são ***declarados public***, mais adiante no livro explicaremos quando usar construtores ***private***.

**3.6 - Construtor padrão**

- Lembre-se de que a linha 13 da Figura 3.2 - ***Account myAccount = new Account();***, usou ***new*** para criar um objeto Account. Os ***parênteses vazios*** depois de “***new Account***” indicam uma ***chamada para o construtor padrão*** da classe.

- Em qualquer classe que nãodeclare explicitamente um construtor, o compilador fornece um tipo padrão (sempre que não tem parâmetros). Quando uma classe tem somente o construtor padrão, as variáveis de instância da classe são inicializadas de acordo com seus valores padrões. Na Seção 8.5, você aprenderá que as classes podem ter múltiplos construtores.

- ***Não há nenhum construtor padrão em uma classe que declara um construtor***

Se você declarar um construtor para uma classe, o compilador não criará um construtor padrão para ela. Nesse caso, você não será capaz de estabelecer um objeto Account com a expressão de criação de instância da classe new Account(), como fizemos na Figura 3.2 — a menos que o construtor personalizado que você declare nãoreceba nenhum parâmetro.

**3.7 - GUI - Graphical User Interface *(Interface Gráfica do Usuário)***

- A GUI serve para criar ***interfaces gráficas do usuário (graphical user interfaces — GUIs)*** utilizando a tecnologia Swing que deverá ser substituída pela nova tecnologia JavaFX, que será mostrada nos capítulos posteriores.

**3.8 - Exibindo texto em uma caixa de diálogo**

- Os programas apresentados até agora exibem a ***saída*** na ***janela de comando***. Muitos aplicativos utilizam ***janelas*** ou ***caixas de diálogo*** (também chamadas diálogos) ***para exibir a saída***.

- ***Navegadores web*** como o ***Chrome***, ***Firefox***, ***Internet Explorer***, ***Safari*** e ***Opera*** apresentam páginas da ***web*** em ***janelas*** próprias.

- Os ***programas de correio eletrônico*** permitem digitar e ler mensagens em uma ***janela***.

- Tipicamente, as ***caixas de diálogo*** são ***janelas*** nas quais os ***programas mostram mensagens*** importantes aos ***usuários***.

- A ***classe JOptionPane*** fornece caixas de diálogo pré-construídas que permitem aos programas exibir janelas que contêm mensagens, essas janelas são chamadas de mensagens de diálogo.

**- O Programa utilizando a Classe JOptionPane**

*package livrojcp\_cap3\_ex09\_joptionpane;*

*// Usando JOptionPane para exibir múltiplas linhas em uma caixa de diálogo.*

*import javax.swing.JOptionPane;*

*public class LivroJCP\_Cap3\_Ex09\_JOptionPane {*

*public static void main(String[] args) {*

*/\* Chama o método JOptionPane showMessageDialog o qual possui 02 argumentos*

*o primeiro quando preenchido (no caso null), serve para posicionar a caixa*

*de diálogo na tela e o segundo exibe a string.*

*\*/*

*JOptionPane.showMessageDialog(null, "Bem Vindo ao Java");*

*}*

*}*

**3.9 - Método static showInputDialog da classe JOptionPan**

- O método **s*howInputDialog*** de **J*OptionPane*** exibe uma caixa de diálogo de ***entrada*** que contém um prompt e um campo (conhecido como campo de texto) no qual o usuário pode inserir o texto.

- O ***argumento*** do ***método*** showInputDialog é o ***prompt*** que espera o usuário digitar uma string.

- O ***método*** showInputDialog ***retorna*** uma ***String*** contendo os caracteres digitados pelo usuário.

- Se você ***pressionar*** o botão ***Cancel*** na caixa de diálogo ou a ***tecla Esc***, o método ***retornará*** ***null*** e o programa exibirá a palavra ***“null”*** como nome.

**Capítulo-4 - Instruções de controle: parte 1; 4 operadores de atribuição ++ e --**

**4.1 - GUIs e Imagens Gráficas: Criando Desenhos Simples - *Sistema de Coordenadas do Java***

- Por padrão, o ***canto superior esquerdo*** de um componente da ***GUI*** tem as ***coordenadas (0, 0).***

- Um ***par*** ***de*** ***coordenadas*** é composto de uma ***coordenada*** ***x*** (a coordenada horizontal) e uma ***coordenada*** ***y*** (a coordenada vertical).

- A coordenada x é a localização horizontal que se estende ***da esquerda para a direita.***

- A coordenada y é a localização vertical que se estende ***de cima para baixo***.

- O eixo x descreve cada coordenada horizontal e o eixo y, cada coordenada vertical.

- As ***coordenadas indicam*** onde elementos ***gráficos*** devem ser ***exibidos*** em uma ***tela***.

- Unidades coordenadas são ***medidas em pixels***. O termo ***pixel*** significa ***"picture element"*** *(elemento de imagem).* Um pixel é a ***menor unidade*** de exibição da ***resolução do monitor***.

**4.2 - Primeiro aplicativo de desenho**

- Nosso primeiro ***aplicativo*** de desenho simplesmente ***desenha duas linhas***.

- A ***classe DesenhaPainel*** *(DrawPanel - Figura 4.18 do livro)* realiza o ***desenho real***.

- A ***classe Cap4\_01\_DesenhaTeste*** *(classe DrawPanelTest - Figura 4.19 do livro)* cria uma janela para exibir o desenho.

- Na ***classe DesenhaPainel***, as instruções import nas ***linhas 3 a 4*** permitem utilizar a classe ***Graphics*** (do ***pacote java.awt***), que ***fornece*** vários ***métodos*** para desenhar texto e formas na tela.

- A classe ***JPanel*** do pacote ***javax.swing***, fornece uma ***área em que podemos desenhar.***

**- A classe *Cap4\_01\_DesenhaTeste***

*package cap4\_01\_desenhateste;*

*import javax.swing.JFrame;*

*//A classe 'Cap4\_01\_DesenhaTeste' CRIA uma janela para exibir o desenho*

*public class Cap4\_01\_DesenhaTeste {*

*public static void main(String[] args) {*

*// cria um objeto janela que contém nosso desenho*

*DesenhaPainel janela = new DesenhaPainel(); // janela = panel do livro*

*// Cria o objeto 'quadro da classe JFrame e exige importação do 'import javax.swing.JFrame;'.*

*JFrame quadro = new JFrame();*

*// configura o Frame para ser encerrado quando ele é fechado*

*quadro.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);*

*quadro.add(janela); // Adiciona o quadro ao Frame.*

*// Configura o tamanho da janelapassando os parãmetros para as variáveis ‘largura e altura’*

*quadro.setSize(600, 400);*

*quadro.setVisible(true); // Torna o Frame visível.*

*}*

*} // Fim da class Cap4\_01\_DesenhaTeste*

**- A classe *DesenhaPainel extends JPanel***

*package cap4\_01\_desenhateste;*

*import java.awt.Graphics;*

*import javax.swing.JPanel;*

*//A classe 'DesenhaPainel EXECUTA o desenho das linhas.*

*public class DesenhaPainel extends JPanel { //Exige importação do 'import javax.swing.JPanel;'*

*// desenha um X a partir dos cantos do painel e exige importação do 'import java.awt.Graphics;'*

*//paintComponent NÃO pode ser alterado*

*public void paintComponent(Graphics desenhaLinha) {*

*// chama paintComponent para assegurar que o painel é exibido corretamente*

*super.paintComponent(desenhaLinha);*

*/\*Como 'DesenhaPainel estende (extends) JPanel, então 'DesenhaPainel' pode utilizar alguns métodos public de JPanel. Os métodos getWidth e getHeight retornam a largura e a altura de JPANEL, respectivamente*

*\*/*

*int largura = getWidth(); //Largura total (width).*

*int altura = getHeight(); //Altura total (altura).*

*// desenha uma desenhaLinha a partir do canto superior esquerdo até o inferior direito*

*desenhaLinha.drawLine(0, 0, largura, altura);*

*// desenha uma desenhaLinha a partir do canto inferior esquerdo até o superior direito*

*desenhaLinha.drawLine(0, altura, largura, 0);*

*}*

*}*

**Capítulo-5 - Instruções de Controle: parte 2; Operadores Lógicos**

**5.1 - Anatomia do Comando System.out.print**

- ***System.out*** . 🡪 é um ***objeto*** da classe ***java.lang.System***

- ***printf*** 🡪 é o ***método*** do ***objeto*** System.out

- ***println*** ­­🡪 é o ***método*** do ***objeto*** System.out

**5.2 - Programa para Calcular Juros Compostos**

**- A classe *Cap5\_01\_JurosCompostos***

***package cap5\_01\_juroscompostos;***

***public class Cap5\_01\_JurosCompostos {***

***/\* vf = va \* (1 + txj)^a***

***onde***

***vf = Valor futuro no fim do n-ésimo ano.***

***va = valor atual original investida (isto é, o valorAtual)***

***txj = é a taxa de juros anual (por exemplo, utilize 0,05 para 5%)***

***a = é o número de anos***

***\*/***

***public static void main(String[] args) {***

***double valorFuturo; // Valor futuro***

***double valorAtual = 1000.00; // Valor inicial***

***double taxaJuros = 0.05; // Taxa de juros anual***

***//double taxaJuros = 0.0041666667; // Taxa de juros mensal***

***// Exibe o cabeçalho***

***System.out.printf("%s%17s %n", "Ano", "Valor Futuro");***

***//Calcula os depósitos para cada um dos 10 anos.***

***for (int ano = 1; ano <= 10; ++ano){***

***//Calcula nova quantidade durante ano especificado.***

***valorFuturo = valorAtual \* Math.pow(1.0 + taxaJuros, ano);***

***/\* A máscara %2d, exibe o valor do ano abaixo da letra 'n' de 'ano'.***

***\* A máscara %,16.2f exibe o Valor Futuro, onde a vírgula ',' após o '%'***

***\* serve para indicar que o valor deve ser exibido com um ponto separando***

***\* cada grupo dos milhares, milhões, bilhões, etc.s (ex. 1.647,01) e o '2'***

***\* antes do 'f' indica a qtde de casas decimais a serem exibidas.***

***\* O '16' indica quantas espaços em branco entre os valores de 'ano' e 'V.Futuro'.***

***\*/***

***System.out.printf("%2d%16.2f %n", ano, valorFuturo); // A máscara %2d,***

***}***

***}***

***}***

**5.2 - Formatando números de ponto flutuante**

- A máscara ***%2d***  exibe o ***valor*** do ***ano*** abaixo da letra ***'n'*** de ***'ano'.***

- A máscara ***%,16.2f*** exibe o ***Valor Futuro***, onde a ***vírgula ','*** após o '***%***' serve para indicar que o valor deve ser exibido com um ***ponto*** separando cada ***grupo dos milhares***, ***milhões, bilhões,*** etc.s (ex. 1.647,01) e o ***'2'***  antes do ***'f'*** indica a qtde de ***casas decimais*** a serem ***exibidas***.

- O ***'16'*** indica quantos ***espaços em branco*** deve ser colodado entre os valores ***de 'ano'*** e ***'V.Futuro'***.

**Capítulo-6 - Métodos: um exame mais profundo**

***6.1 Módulos de programa em Java***

- Você escreve programas Java combinando novos métodos e classes com aqueles predefinidos disponíveis na ***Java Application Programming Interface*** (também chamada ***Java API*** ou ***biblioteca de classes Java***) e em várias outras bibliotecas de classes.

- Classes relacionadas são agrupadas em ***pacotes***de modo que possam ser ***importadas***nos programas e ***reutilizadas***.

- Você aprenderá a agrupar suas próprias classes em pacotes na Seção 21.4.10.

- A ***Java*** ***API*** fornece uma rica ***coleção de classes*** predefinidas que contém métodos para realizar:

1 ***- cálculos matemáticos comuns,***

***2 - manipulações de string,***

***3 - manipulações de caractere,***

***4 - operações de entrada/saída,***

***5 - operações de banco de dados,***

***6 - operações de rede,***

***7 - processamento de arquivo, verificação de erros etc.***

***Observação de engenharia de software***

- Familiarize-se com a rica coleção de classes e métodos fornecidos pela Java API

***http://docs.oracle.com/javase/7/docs/api/).***

- A Seção 6.8 fornece uma visão geral dos vários pacotes comuns.

- O Apêndice, em inglês, na Sala Virtual do livro, explica como navegar pela documentação da API.

- Não reinvente a roda. Quando possível, reutilize as classes e métodos na Java API. Isso reduz o tempo de desenvolvimento de programas e evita a introdução de erros.

***Dividir para conquistar com classes e métodos***

- Classes e métodos ajudam a modularizar um programa separando suas tarefas em unidades autocontidas. - As instruções no corpo dos métodos são escritas apenas uma vez, permanecem ocultas de outros métodos e podem ser reutilizadas a partir de várias localizações em um programa. Uma motivação para modularizar um programa em métodos e classes é a abordagem *dividir para conquistar*, que torna o desenvolvimento de programas mais gerenciável, construindo programas a partir de peças mais simples e menores. Outra é a capacidade de reutilização de software — o uso de classes e métodos existentes como blocos de construção para criar novos programas.

***6.2 Métodos static, campos static e classe Math***

***- Método static ou método de classe*** 🡪 É quando um método realiza uma tarefa que não depende de um objeto específico, ou seja, esse método se aplica à classe em que é criado.

***6.3 As palavras-chave public e static***

- A ***palavra-chave*** ***public*** indica que o método pode ser chamado a partir dos métodos das outras classes.

- A ***palavra-chave*** ***static*** permite que o método ***main*** (outro método static) chame métodos dentro da mesma classe sem qualificar o nome do método com o nome da classe.

- ***Métodos static*** na mesma classe podem ***chamar*** uns aos outros diretamente, porém qualquer outra classe que usar tais métodos deve ***qualificar*** ***totalmente*** o nome do método com o ***nome da classe***.

***Observação de engenharia de software***

- ***Métodos*** podem retornar ***no máximo um valor***, mas o valor retornado ***poderia*** ser uma ***referência*** a um ***objeto*** que contém ***muitos valores***.

***6.4 Escopo*** *(alcance)* ***das declarações***

- Você viu ***declarações*** de várias ***entidades*** ***Java*** como ***classes***, ***métodos***, ***variáveis*** e ***parâmetros***.

- As declarações introduzem nomes que podem ser utilizados para referenciar essas entidades Java.

- O ***escopo*** *(alcance)* de uma declaração é a parte do programa que pode referenciar a entidade declarada pelo seu nome.

- Diz-se que essa ***entidade*** está no ***escopo*** *(no alcance)* para essa parte do programa.

***- As regras básicas de escopo são estas:***

1. O ***escopo*** de uma ***declaração*** de ***parâmetro*** é o ***corpo do método*** em que a declaração aparece.

2. O ***escopo*** de uma ***declaração de variável*** ***local*** é do ponto em que a declaração aparece ***até o final desse bloco.***

3. O ***escopo*** de uma ***declaração de variável local*** que aparece na seção de inicialização do ***cabeçalho de uma instrução for*** é o corpo da instrução ***for*** e as outras expressões no cabeçalho.

4. O ***escopo*** de um ***método ou campo*** é o ***corpo inteiro da classe***. Isso permite que os métodos de instância de uma classe usem os campos e outros métodos da classe.

- Qualquer ***bloco*** pode conter ***declarações de variável***.

- Se uma ***variável local*** ou um ***parâmetro*** em um ***método*** tiver o ***mesmo nome*** de um ***campo da classe***, o campo permanece ***oculto*** até que o ***bloco termine*** a execução — isso é chamado de ***sombreamento***.

- ***Para acessar um campo sombreado em um bloco:***

• Se o ***campo*** é uma ***variável de instância***, preceda o nome com a ***palavra-chave this e um ponto (.),*** como em ***this.x.***

• Se o campo é uma ***variável de classe static***, ***preceda*** o nome com o ***nome da classe e um ponto (.)***, como em NomeDaClasse.x.

- ***O programa abaixo demonstra os problemas de escopo com campos e variáveis locais:***

🡪A linha 7 declara e inicializa o campo x para 1. Esse campo permanece ***sombreado (oculto)*** em qualquer ***bloco*** ***(ou método)*** que ***declara*** uma ***variável local chamada x***.

🡪 O ***método main*** (linhas 11 a 23) ***declara*** uma ***variável local x*** (linha 13) e a ***inicializa*** para ***5***. O valor dessa variável local é gerado para mostrar que o campo x (cujo valor é 1) permanece sombreado no método main.

🡪O programa declara outros dois métodos — *usaVariavelLocale* (linhas 26 a 35) e *usaCampo* (linhas 38 a 45) — que não recebem argumentos e não retornam resultados.

🡪 O método main chama cada método duas vezes (linhas 17 a 20). O método *usaVariavelLocale* declara a variável local x (linha 28). Quando useLocalVariable é chamado pela primeira vez (linha 17), ele cria a variável local x e a inicializa como 25 (linha 28), gera a saída do valor de x (linhas 30 e 31), incrementa x (linha 32) e gera a saída do valor de x novamente (linhas 33 e 34).

- Quando *usaVariavelLocale* é chamado uma segunda vez (linha 19), ele recria a variável local x e a reinicializa como 25, assim a saída de cada chamada a *usaVariavelLocale* é idêntica.

**- A classe cap6\_05\_escopovariaveis**

***package cap6\_05\_escopovariaveis;***

***public class Cap6\_05\_EscopoVariaveis {***

***// o 'private static int x = 1' é um Campo acessivel para todos os métodos dessa classe***

***private static int x = 1;***

***// O método main cria e inicializa a variável local 'x'***

***// e chama os métodos 'usaVariavelLocal' e 'usaCampo'***

***public static void main(String[] args) {***

***int x = 5; // Esta Variável local 'x' do método sombreia o campo 'x'.***

***//System.out.printf("Variável local 'x' no método main é = %d%n", x);***

***System.out.println("Variável local 'x' do método main é = " + x);***

***System.out.println("----------------------------------------------------");***

***usaVariavelLocal(); // Este método tem uma variável local 'x'.***

***System.out.println("---------------------------------------------------");***

***usaCampo(); // Este método utiliza o campo 'x' da classe 'Cap6\_05\_EscopoVariaveis'.***

***System.out.println("----------------------------------------------------");***

***usaVariavelLocal(); // Nesta chamada este método reinicializa a variável local 'x'.***

***System.out.println("SCOPO LOCAL VARIÁVEL DO MÉTODO 'usaVariavelLocal' NÃO ALTERA");***

***System.out.println("----------------------------------------------------");***

***usaCampo(); // Nesta chamada o método mantém o valor do campos 'x'.***

***System.out.println("----------------------------------------------------");***

***//System.out.printf("%nVariável local 'x' no método main é %d%n", x);***

***System.out.println("SCOPO LOCAL VARIÁVEL DO MÉTODO 'main' NÃO ALTERA");***

***System.out.println("Variável local 'x' do método main é = " + x);***

***System.out.println("----------------------------------------------------");***

***}***

***// Cria e inicializa a variável local 'x' durante cada chamado.***

***public static void usaVariavelLocal() {***

***int x = 25; // Inicializa toda vez que este método é chamado.***

***//System.out.printf("%nVariável Local 'x' ao entrar no método 'usaVariavelLocal' é = %d%n", x);***

***System.out.println("Método 'usaVariavelLocal': 'x' = " + x);***

***++x; // Incrementa a variável local 'x' no método 'usaVariavelLocal'.***

***//System.out.printf("Variável local 'x' incrementada do valor 1 dentro do método 'usaVariavelLocal' é = %d%n", x);***

***System.out.println("Método 'usaVariavelLocal': 'x' = 25+1 = " + x);***

***//System.out.println("----------------------------------------------------");***

***}***

***// Modifica o campo 'x' da classe 'Cap6\_05\_EscopoVariaveis' durante cada chamada do método 'usaCampo'.***

***public static void usaCampo() {***

***//System.out.printf("%nValor do campo 'x' ao entrar no método 'usaCampo' é = %d%n", x);***

***//x \*= 10; // Multiplica por 10 o campo 'x' da classe 'Cap6\_05\_EscopoVariaveis'.***

***System.out.println("Método 'usaCampo': campo 'x' = " + x);***

***x \*= 10; // Multiplica por 10 o campo 'x' da classe 'Cap6\_05\_EscopoVariaveis'.***

***//System.out.printf("Valor do campo 'x' antes de sair do método 'usaCampo' é = %d%n", x);***

***System.out.println("Método 'usaCampo': O campo 'x' = x \* 10 = " + x);***

***//System.out.println("----------------------------------------------------");***

***}***

***}***

***6.5 Sobrecarga de método*** *(Overload)*

- Um ***método*** com mesmo ***nome*** pode ser declarado ***várias*** ***vezes*** na ***classe***, contanto que tenha uma ***assinatura diferente*** para o conjunto de parâmetros (***determinados*** pela ***quantidade***, ***tipos*** e ***ordem dos parâmetros***). O Java não considera o tipo de ***retorno***.

- Quando um método sobrecarregado é chamado, o ***compilador Java seleciona*** o método adequado examinando a ***quantidade***, os ***tipos*** e a ***ordem*** dos ***argumentos*** na chamada.

- A sobrecarga de métodos é comumente utilizada para criar vários métodos com o mesmo nome que realizam as mesmas tarefas, ou tarefas semelhantes, mas sobre ***tipos diferentes*** ou ***quantidades diferentes*** de argumentos.

O exercício abaixo chama o ***método*** ***sobrecarga***() 5 vezes:

1 - sobrecarga(***int***) -

2 - sobrecarga(***int***, ***int***) - varia a qtde

3 - sobrecarga(***double***) - varia o tipo de parâmetro

4 - sobrecarga(***double***, ***float***) - varia o tipo e a qtde

5 - sobrecarga(***float***, ***double***) - varia a qtde

**- A classe cap6\_06\_metodosSobrecarregados**

***package cap6\_06\_metodossobrecarregados;***

***/\* SOBRECARGA DE MÉTODOS (OVERLOAD)***

***\* Métodos com mesmo nome podem ser declaradom várias vezes na classe, contanto que tenham uma***

***\* assinatura diferente para o conjunto de parâmetros a qual é determinada pela quantidade,***

***\* tipos e ordem dos parâmetros.***

***\* O Java não considera o tipo de retorno.***

***\****

***\* Este exercício chama o método 'sobrecarga() 5 vezes:***

***\* 1 - sobrecarga(int) -***

***\* 2 - sobrecarga(int, int) - varia a qtde***

***\* 3 - sobrecarga(double) - varia o tipo de parâmetro***

***\* 4 - sobrecarga(double, float) - varia o tipo e a qtde***

***\* 5 - sobrecarga(float, double) - varia a qtde***

***\*/***

***public class Cap6\_06\_MetodosSobreCarregados {***

***public static void main(String[] args) {***

***// Chama o método 'sobrecarga' passando um parâmetro tipo 'inteiro'.***

***System.out.printf("O quadrado do n. inteiro 7 é: %d%n", sobrecarga(7));***

***// Chama o método 'sobrecarga' passando dois parâmetros tipo 'inteiro'.***

***System.out.printf("O produto do ns. inteiros 4x6 = %d%n", metodoQuadrado(4, 6));***

***// Chama o método 'sobrecarga' passando um parâmetro tipo 'double'.***

***System.out.printf("O quadrado do n. double 7.5 é: %f%n", metodoQuadrado(7.5));***

***// Chama o método 'sobrecarga' passando dois parâmetros tipo 'double e float'.***

***System.out.printf("A soma do n. double e float 7.5+2.5 = %f%n", metodoQuadrado(7.5, 2.5F));***

***// Chama o método 'sobrecarga' passando dois parâmetros tipo 'float e double'.***

***System.out.printf("A subtração dos ns. float e double 3.5-4.5 = %f%n", metodoQuadrado(3.5f, 4.5));***

***}***

***// Método 'sobrecarga' com um argumento tipo int.***

***public static int sobrecarga(int intValue) {***

***System.out.printf("%nChamada do 'metodoQuadrado' com argumento inteiro: %d%n", intValue);***

***return intValue \* intValue;***

***}***

***// Método 'sobrecarga' com dois argumentos tipo int.***

***public static int metodoQuadrado(int intValue1, int intValue2) {***

***System.out.printf("%nChamada do 'metodoQuadrado' com argumentos inteiros: %d e %d%n", intValue1, intValue2);***

***return (intValue1 \* intValue2);***

***}***

***// Método 'sobrecarga' com argumento tipo double.***

***public static double metodoQuadrado(double doubleValue) {***

***System.out.printf("%nChamada do 'metodoQuadrado' com argumento double: %f%n", doubleValue);***

***return doubleValue \* doubleValue;***

***}***

***// Método 'sobrecarga' com dois argumentos double e float.***

***public static double metodoQuadrado(double doubleValue, float floatValue) {***

***System.out.printf("%nChamada do 'metodoQuadrado' com argumentos double e float: %f %f%n", doubleValue, floatValue);***

***return doubleValue + floatValue;***

***}***

***// Método 'sobrecarga' com dois argumentos float e double.***

***public static double metodoQuadrado(float floatValue, double doubleValue) {***

***System.out.printf("%nChamada do 'metodoQuadrado' com argumentos float e double: %f %f%n", floatValue, doubleValue);***

***return floatValue - doubleValue;***

***}***

***}***