**Nome:** Guilherme Fabiano Terra da Silva

**CURSO DE JAVA NA UDEMY**

1. **SEÇÃO 1**
2. **Objetivo:** Aprender Java e programação orientada a objetos;
3. **SEÇÃO 2**
4. **Algoritmo:** É uma sequência de instruções que solucionam um problema. Não necessariamente está ligado a programação;
5. **Automação:** Utilizar de uma máquina pra realizar um processo desejado, tornando-o automático ou semiautomático;
6. **Computador**

* **Hardware** (Parte física);
* **Software** (Programas);
* É uma máquina que automatiza os algoritmos;

1. **O que é necessário para se fazer um programa?**

* **Linguagem de programação:** São um conjunto de regras para se desenvolver um programar. OBS: Regras Léxicas + Sintáticas;
* **IDE:** É um software usado para construir / editar e testar seu programa;
* **Compilador:** É um software que **transforma o código fonte em código objeto**, para poder ser executado;
* **Gerador de código ou Máquina Virtual:** Software que permite que o programa obj. seja executado **(no Java é VM – Virtual Machine)**;

1. **Regras Léxicas e Sintáticas**

* **Léxica:** Ortografia – É a maneira que cada palavra deve ser na linguagem;
* **Sintática:** Gramática – É a ordem das palavras e como são usadas;

1. **Código fonte**

* É o código escrito pelo programador em uma linguagem de programação;
* O computador não entende este código;
* Precisa-se de um Compilador para torna-lo legível à máquina;

1. **Compilador**

* O compilador converte o código fonte em **código objeto (.o)**, realizando anteriormente a análise Léxica e Sintática;
* Se estiver tudo correto o código objeto é gerado para ser interpretado por uma VM ou gerador de código;
* Se não, ocorre erros e o desenvolvedor deve resolvê-los;

1. **VM ou gerador de código**

* Pega o código objeto e realiza o processo de construção de um **código executável (.exe)**;
* O processo de construção se chama build;

1. **Interpelação**

* Quando um interpretador lê o código fonte e realiza a função de compilador e VM sob demanda (gradualmente o código fonte é lido e convertido em executável);

1. **Abordagem Híbrida**

* Quando Código fonte passa por **pre-compilação**, gerando **byte code (.class**). Este byte code é executado sobre demanda (gradualmente) por uma VM;
* Java;

1. **SEÇÃO 3**
2. **Instalação do Java e Eclipse (IDE de desenvolvimento);**
3. **Versão LTS:** É uma versão de longo tempo de suporte, mais usada pelas empresas para não haver necessidade de atualizações de versões com alta velocidade;
4. **Classes**

* São as entidades que possuirão atributos. **Ex:** Clientes, Produtos e etc;

1. **Packages**

* Agrupamento de classes relacionadas;

1. **Módulos**

* Agrupamento de pacotes relacionados. **Ex:** Financeiro, produtivo, marketing e etc;

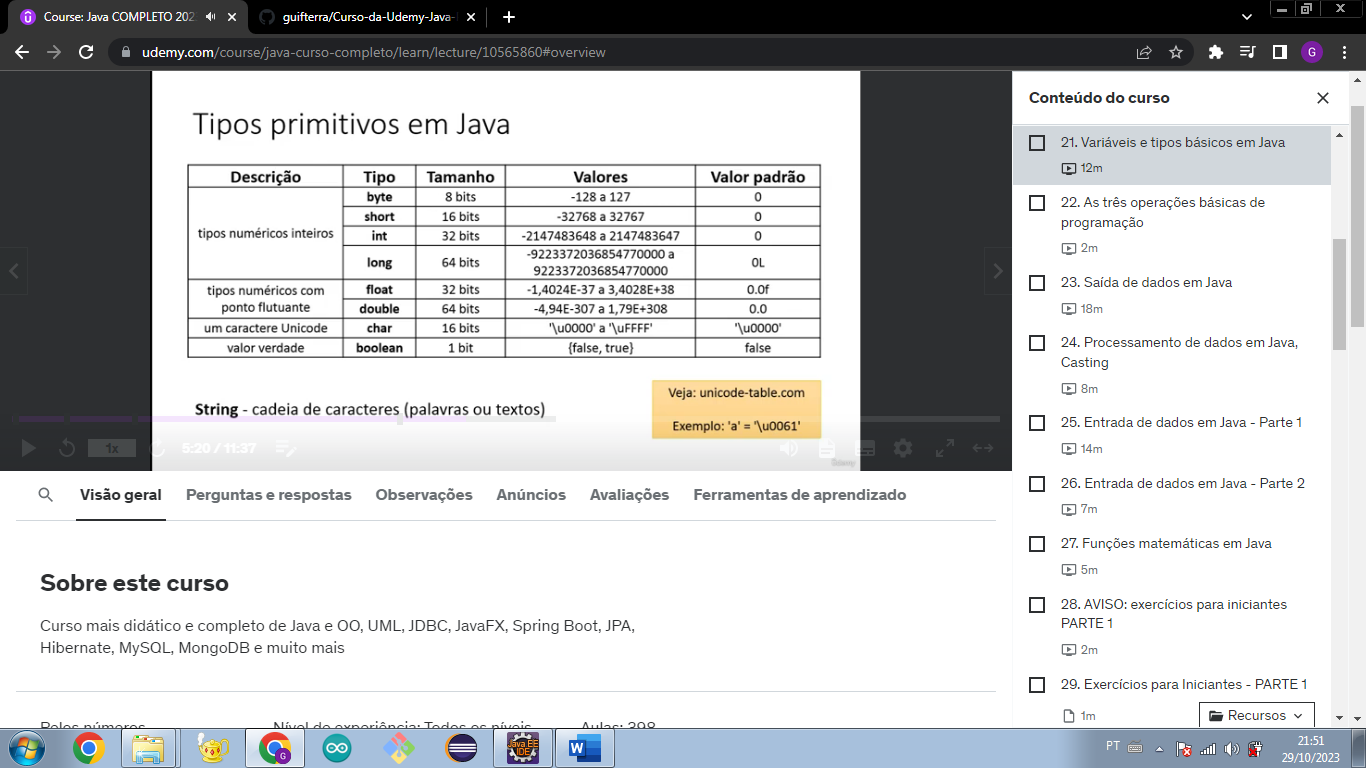
1. **Aplicação**

* Agrupamento dos módulos;

1. **SEÇÃO 4**
2. **Operadores aritméticos:** Soma (+), Subtração (-), Divisão (/), Multiplicação (\*) e Módulo (%);
3. **Variáveis:** Porções da memória RAM que armazenam dados;
4. **Declaração:**

* Sintaxe: <tipo> <nome / identificador> = <valor>;
* **Toda variável possui endereço**, pois ela está alocada em algum lugar da memória;

1. **Tipos primitivos**



1. **Impressões**

* System.out.print;
* System.out.println;
* System.out.printf;

1. **Processamento de dados**

* Realizado pelo operador de atribuição (=);

1. **Casting**

* É a conversão de tipos dos dados;

1. **Leitura de dados**

* Uso do Scanner;
* Scanner <nome> = new Scanner (System.in);

1. **SEÇÃO**
2. **Expressões comparativas**

* ==, >, <, <=, >=, != ;

1. **Expressões lógias**

* **&& :** And ou ‘E’;
* **|| :** Or ou ‘OU’;
* **! :** Not ou ‘NÃO’;

1. **Estrutura condicional**

* If ( <condição> ) { <comandos> } else { <comandos> };

1. **Operadores de atribuição cumulativa**

* +=
* -=
* \*=
* /=
* %=

1. **Switch case**

* Switch( <condição>){ case <valor> : <código> break; ... }
* Pode ter um default (Caso nenhum case seja realizado);

1. **Expressão condicional ternária**

* ( <condição> )? <valor> : <valor> ;

1. **SEÇÃO 6**
2. **Estruturas de repetição**

* While (<condição>) { <códigos> };
* For ( <inicio>; <condição>; <incremento> ) { <códigos> };
* Do { <código> } While( <condição> );

1. **SEÇÃO 7**
2. **Nomes de variáveis**

* Não pode começar com número;
* Sem acento;
* Padrão camel case;
* Classe começa com nome maiúsculo;
* Atributos, variáveis e etc – Começam com nome minúsculo;

1. **Funções para String**

* **Formatar:** toLowerCase(), toUpperCase() e trim();
* **Recortar:** substring(<inicio>,<fim>);
* **Substituir:** replace(<”texto”>,<”texto”>);
* **Buscar:** IndexOf(), LastIndexOf();

1. **Funções**

* Processamento que tem significado e pode ser reaproveitada várias vezes;

1. **Seção 8 – Inicio da POO**
2. **Classe**

* **É uma abstração de objeto, sendo um modelo, uma forma de bolo para objetos;**
* É um tipo estruturado / personalizado;
* Pode conter atributos e métodos (TAMBÉM CHAMADOS MEMBROS);

1. **Instanciação**

* A memória possui áreas, no caso atual, são elas: **Stack** e **Heap**;
* **Stack** é onde são criadas variáveis estáticas;
* Ao se usar o comando **new nomeDaClasse()** cria-se uma instância do Objeto do tipo nomeDaClasse na área Heap da memória;
* No **Stack** fica o **endereço do objeto criado lá no Heap**;

1. **Pra que servem os métodos dentro das classes?**

* Reaproveitamento de código (Não repetir);
* Delegação de responsabilidade ou Coesão (A classe X é responsável pelos cálculos ou processos dos obj. tipo X);

1. **Diagrama UML**

|  |
| --- |
| **Nome da classe** |
| Atributos  ou  Característica |
| **Métodos da classe** |

1. **This**

* Informa no código que a referência é para o atributo;

1. **String.format**

* Para formatar uma var sem usar o printf
* Formato: String.format ( <”%formatação”>, <nomeDaVar> );

1. **toString na entidade**

* Ao se criar o método to String na classe da entidade, você pode editar a saída do objeto no momento de impressão;
* Permite impressões personalizadas;

1. **Membros estáticos**

* Não precisam de objetos para ser chamados;
* São chamados pelo nome da própria classe;
* Dentro de uma classe estática, não se pode usar / chamar outras classes não estáticas sem criar um objeto. Ou seja, para usar uma classe não estática dentro de um contexto estático, deve-se criar um objeto do tipo **classeQueDesejaUsar** para ter acesso as funções;
* Todo programa que EU fiz até o momento, tem o: **public static void main(String args[])**, um contexto estático. Nele, eu só posso chamar outras classes não estáticas, se não é estática deve-se criar um objeto para acessar;
* Para chamar usa-se:

- NomeDaClasse.variável

- NomeDaClasse.método

1. **Declaração de constantes**

* Uso da palavra final;
* Por padrão, constantes recebem nomes todos maiúsculos;
* Palavras separadas por \_

1. **Construtores, palavra this e encapsulamento**
2. **Construtor**

* É uma operação da classe (método) que é executada quando o objeto é instanciado (ao se usar o new);
* Serve para dar valores iniciais para os atributos ou permitir / obrigar que dados sejam fornecidos no momento de instanciar o objeto;
* Caracterizado pela estrutura: public nomeDaClasse() { }

1. **This**

* É uma referência para o próprio objeto;
* Diferencia o **atributo do objeto** de **variáveis locais**;

1. **Sobrecarga**

* É um recurso que permite uma classe oferecer mais de um método com o mesmo nome, mas com assinaturas diferentes;
* Por assinatura quero dizer a lista de parâmetros;

1. **Encapsulamento**

* Esconder detalhes da implementação, permitindo que o usuário tenha acesso somente a operações seguras e validadas, que mantenham a consistência da classe;
* Os atributos devem ser privados e permitir seu acesso somente a partir dos métodos gets e sets;

1. **Modificadores de acesso**

* **Private:** Só pode ser acessado na própria classe;
* **(nada):** Só pode ser acessado na classe do mesmo pacote;
* **Protected:** Só pode ser acessado no mesmo pacote e em subclasses de pacotes diferentes;
* **Public:** Pode ser acessado em outras classes, a não ser que esteja em outro módulo;

1. **Memória, Arrays e Listas**
2. **Tipos referência**

* Em java, classes não podem ser vistos como caixas que guardam valores, mas sim um ponteiro;
* Na realidade, quando se usa um tipo referência a variável guarda o endereço de memória em que o objeto está alocado;
* A variável criada fica na área da memória **Stack** que aponta para a área em que está o objeto, **Heap**;

1. **Valor nulo**

* Tipos referência aceitam valor nulo (não aponta para ninguém);

1. **Tipos primitivos / valor**

* Tipos primitivos são como caixas na memória que armazenam um valor;
* Elas ficam na área da memória chamada **Stack**;

1. **Valores padrão**

* Quando se cria objetos com new,os seus atributos vem com valores padrão;

1. **Garbage collector**

* É um processo que automatiza o gerenciamento de memória de um programa em execução;
* Ele fica monitorando a área **heap** da memória (onde ficam os objetos) e ele desaloca objetos que não estão sendo utilizados;
* Desaloca objetos não utilizados;
* **Se um objeto no heap perde TODAS as sua referências o garbege collector o desaloca em um futuro próximo;**

1. **Desalocação por escopo**

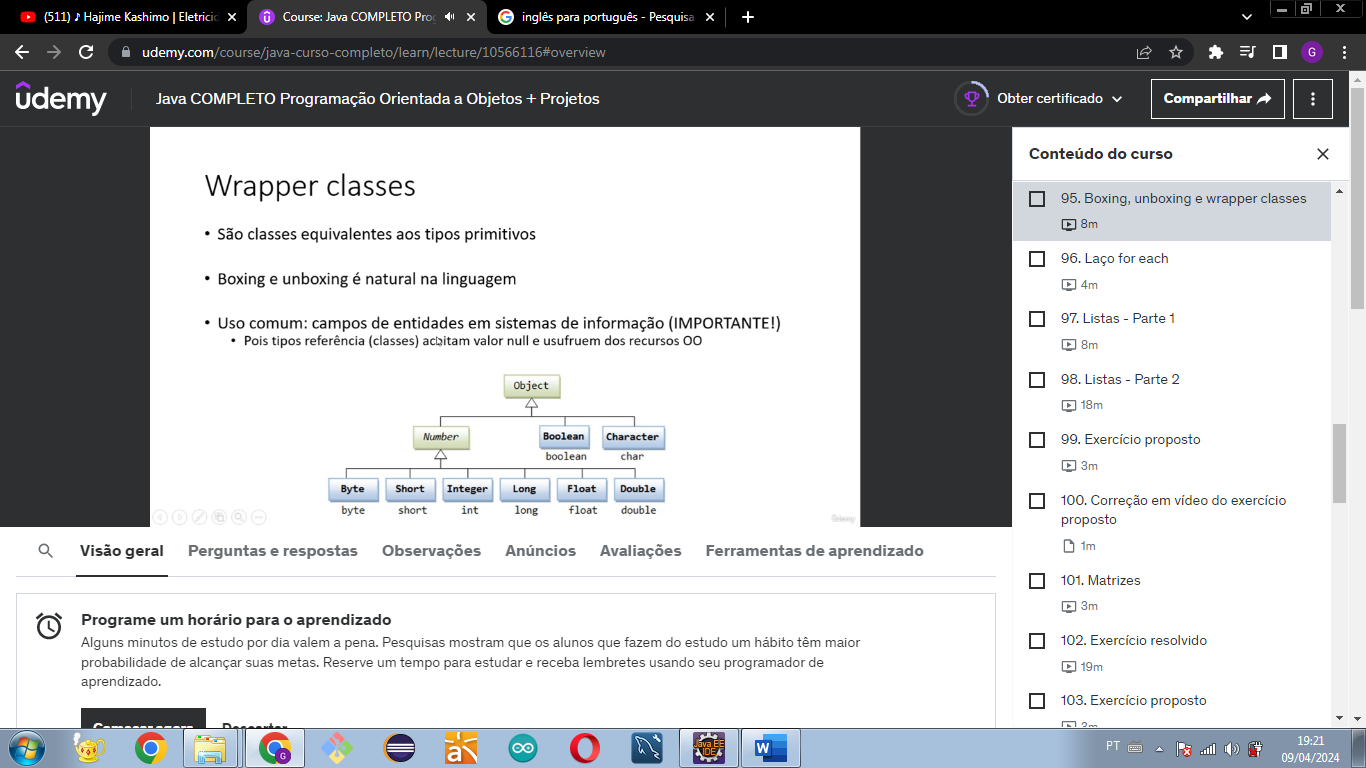
* Garbage collector NÃO É RESPONSÁVEL POR ISSO;

1. **Vetores**

* São arrays unidimensionais;
* São homogêneos (Só aceitam 1 único tipo) e de tamanho fixo;
* Após ser criado, seu tamanho é fixo na memória, dificultando a adesão ou remoção de posições;

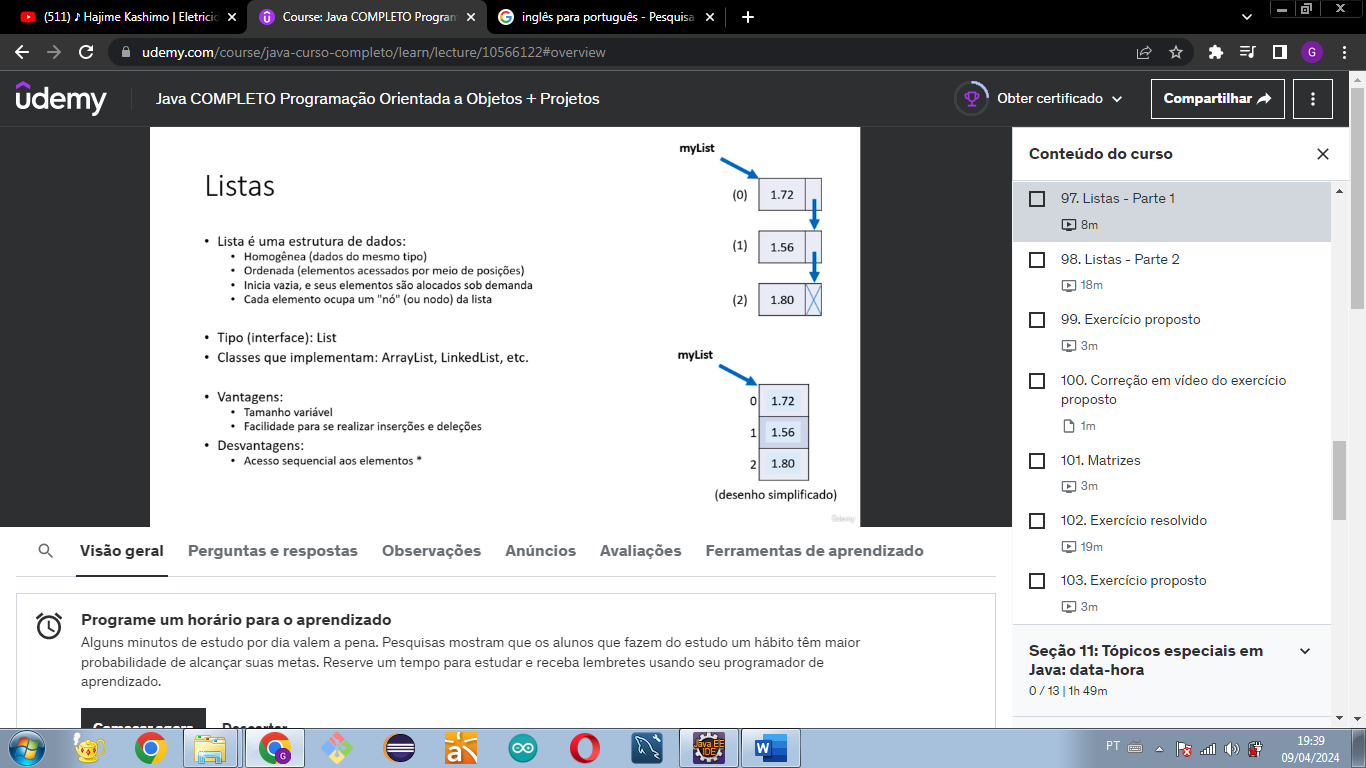
1. **Boxing, Unboxing e Wrapper classes**

* **Boxing:** Converter um tipo valor em objeto;
* **Unboxing:** Converter um tipo objeto em valor;
* **Ex:** int x = 20 - - - - - - - Object obj = x;
* **Ex:** Object obj = 20 - - - - - - - - int y = (int) obj;
* OBS: Wrapper aceitam null, proporcionando maior compatibilidade com BD;



1. **Listas (Em resumo é o vetor de tamanho variavel)**

* Listas são uma estrutura de dados;
* Ela é homogênea (Só aceita 1 único tipo);
* É ordenada (Acessada pelo index);
* Além disso seu tamanho é variável e determinando durante sua utilização, e não em sua declaração;
* **Tipo / Interface:** List;
* **Classes que implementam:** Arraylist, Linkedlist, etc;
* **Funcionamento:** Baseado em nós (também chamados nodos), que são “tipo um vetor de 2 posições”, onde em uma fica o valor e na outra fica o endereço do próximo nó;



1. **Comandos para trabalhar com list**

* size() , add() , remove() , indexOf() , lastIndexOf();
* **Declaração:** List<tipo> nome = new ArrayList<>();
* Lista não aceita tipo primitivo;
* **Add( valor ) ou Add( posição , valor );**
* **Remove( valor ) ou remove( posição );**
* **Removeif( expressão lambda );**
* **Ex:** list.removeIf( x -> x.charAt(0) == ‘M’ );
* Remove todo valor da lista que começa com M;
* **Indexof ( valorBuscado );**
* Retorna -1 se não existe;

1. **Filtro e busca com expressão lambda**

**FILTRO RETORNANDO LISTA:**

list.stream().filter( x -> x.charAt(0) == 'A' ).collect(Collectors.*toList*());

**BUSCA RETORNANDO VALOR:**

list.stream().filter(x -> x.charAt(0) == 'A').findFirst().orElse(**null**);

1. **Matrizes**
2. **DATA E HORA**
3. **Data e Hora**

* Ano-mes-dia-hora;
* **Local:** Sem fuso horário;
* **Usado para ambientes locais;**
* **Global:** Com fuso horário;
* **Usado para ambientes “globalizados”, principalmente na internet;**

1. **Duração**

* Diferença entre duas datas-horas;

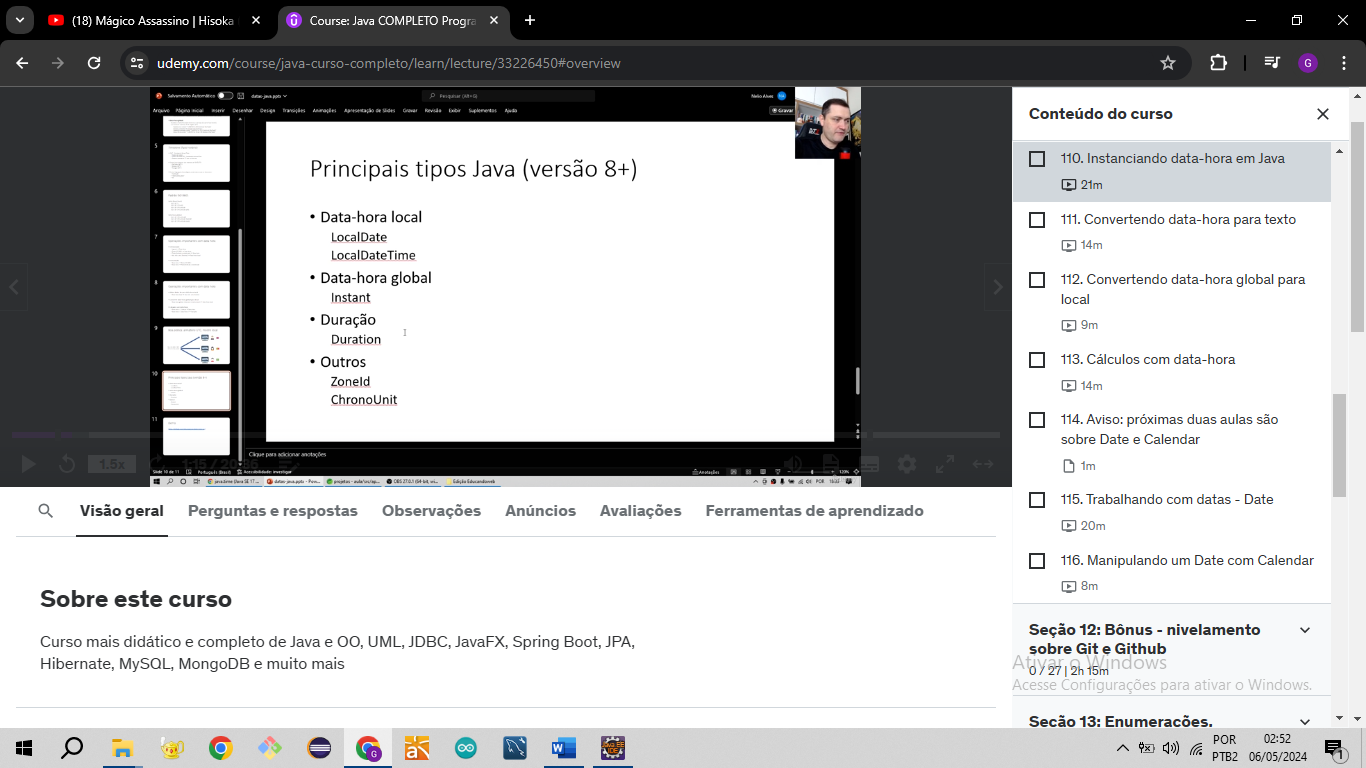
1. **Timezone (fuzo horário)**

* **GMT / UTC (Greenwich Mean Time):** Horário de Londres;
* É o marco 0;
* Também chamado de Z ou Zulu time;

1. **Padrão ISO 0681**

* **Local:** Ano-mês-dia-T-Hora:Minutos:Segundos.Milissegundos;
* **Global:** Ano-mês-dia-T-Hora:Minutos:Segundos.Milissegundos**INDICAÇÃO**;
* **INDICAÇÃO:** Letra ou -XX:00

1. **Data e Hora no Java**



1. **Criando variável de cada tipo**

* NomeDoTipo var;

1. **Pegar instante atual**

* NomeDoTipo.now();

1. **Converter String para um tipo**

* NomeDoTipo.parse(“Texto”);

1. **Formatação de Data e Hora**

* Usa o import java.time.format.DateTimeFormatter;
* Para criar um formato se usa:
* DateTimeFormatter nome = DateTimeFormatter.ofPattern(“fmt”);
* Para aplicar o formato, ao se usar o .parse, se passa o formato;
* LocalDate dt = LocalDate.parse( string , nomeDTF );

1. **Criar DateTime separadamente**

* Uso do of com parâmetros inteiros;
* LocalDate.of( ano , mes , dia );
* Serve para LocalDate, LocalDateTime, ... ;

1. **Passar DateTime para String**

* nomeDT.format( nomeDTF );
* OU
* nomeDTF.format( nomeDT );

**import** java.time.ZoneId;

1. **Passar Instant para String**

* É obrigatório informar o fuso horário;

DateTimeFormatter fmt2 = DateTimeFormatter.*ofPattern*("dd/MM/yyyy HH:mm").withZone(ZoneId.*systemDefault*());

1. **Data e hora global para local**

* LocalDate.ofInstant( Instant , Zone.of(“nome”) );

1. **Pegar dia, mês, ano, hora ... de um LocalDate**

* nomeDaVar.getXXX();

1. **Somar e subtrair algo de uma data**

* nomeData.minus\*\*\*( valor );
* nomeData.plus\*\*\*( valor );

1. **Subtração de datas**

* Duration;
* Duration.between( d1 , d2 );