



# Java

Do básico ao avançado

# Por que aprender Java?

- Alta demanda no mercado de trabalho: Java é uma das linguagens mais populares e procuradas, com muitas oportunidades de emprego;
- Versatilidade: Java é usada em uma ampla gama de aplicações, desde desenvolvimento web e aplicativos móveis até sistemas corporativos;
- Comunidade: Java possui uma comunidade ativa e vasta documentação, facilitando a aprendizagem e a solução de problemas;
- Portabilidade: O código Java é altamente portátil, podendo ser executada em qualquer plataforma que suporte a Java Virtual Machine (JVM);
- Estabilidade e Performance: Java é conhecida por sua robustez, segurança e performance confiável, linguagem amplamente utilizada por grandes empresas;





# Instalando o JDK

- Distribuição de código aberto: Corretto é uma versão gratuita e de código aberto do JDK, mantida pela Amazon;
- Suporte a longo prazo (LTS): Corretto 21 oferece atualizações regulares de segurança e desempenho com suporte garantido pela Amazon;
- Compatível com múltiplas plataformas: Funciona em várias plataformas, incluindo Windows, macOS e Linux.
- Através dele é possível executar o Java de forma fácil na nossa máquina;
- Vamos instalar o Java!



# Instalando o VS Code

- VS Code é um dos melhores editores de código da atualidade;
- Ele permite executar Java de forma simples;
- Utilizaremos também a extensão Extension Pack for Java, que nos ajudará no desenvolvimento de nossas aplicações;
- Vamos lá!





# Primeiro programa

- Para criar nosso primeiro programa vamos precisar de um arquivo .java em uma pasta no nosso computador;
- O código todo fica dentro de uma classe (Class);
- A execução das instruções ficam em um método chamado main;
- Utilizaremos o método println para exibir uma mensagem;
- Através da opção Run conseguimos rodar/executar o programa;
- Vamos lá!



# Considerações gerais

- Java precisa de ponto e vírgula no final de cada linha;
- É uma linguagem case sensitive, ou seja: `println` != `PrintLn`;
- Por ser baseada em orientação a objetos, tem diversos conceitos que utilizaremos de forma abstrata por enquanto (Class, main, System);
- **Não prossiga o curso se o primeiro programa não foi executado corretamente**, abra um chamado para o suporte!





# Executando código pelo terminal

- Primeiramente precisamos compilar o arquivo, utilizando o comando: `javac <nome>;`
- Depois, para executá-lo, temos o comando: `java <nome_arquivo_compilado>;`
- Obs: dependendo da estrutura de pastas a execução pode ficar um pouco diferente (secao1.HelloWorld), e não precisamos colocar a extensão `.java` ou `.class`;
- Isso faz a mesma função de executar no VS Code, e é o processo natural de execução sem uma IDE/Editor de código;
- Note que a cada alteração precisamos compilar o arquivo novamente, vamos lá!



# Código do Curso

- Os arquivos do curso podem ser encontrados em:  
[https://github.com/matheusbattisti/curso\\_java](https://github.com/matheusbattisti/curso_java)
- Faça o download para seguir o curso e ter o código como referência;
- O código está estruturado da mesma maneira passada em aula;







# Introdução

Concluindo a seção



# Variáveis

Introdução



# O que são variáveis?

- Variáveis são espaços na memória que armazenam valores que podem ser manipulados pelo programa;
- Cada variável possui um nome único que identifica o dado armazenado;
- Variáveis têm tipos específicos (como int, String, boolean) que definem o tipo de valor que podem armazenar;
- O valor armazenado em uma variável pode ser alterado durante a execução do programa;
- Variáveis permitem a criação de lógicas dinâmicas, armazenando e manipulando informações ao longo do código;
- Vamos criar uma variável!



# Algumas regras para variáveis

- A sintaxe para declarar variável é: TIPO DE DADO NOME = VALOR;
- O formato de nomes é lowerCamelCase, ou seja: nomeCompleto;
- Os tipos de dados variam conforme a necessidade da variável no programa;
- Cada tipo de dado ocupa um tamanho na memória (tabela na próxima aula), devemos sempre otimizar isso;
- O nome precisa ser único, não pode começar com números;
- Regra para otimização: escolha sempre a que ocupa menos memória;
- O sinal de igual é chamado de atribuição;





# Tipos de dados

- Veja abaixo os tipos de dados em Java:

DATA TYPES	SIZE	DEFAULT	EXPLANATION
boolean	1 bit	false	Stores true or false values
byte	1 byte/ 8bits	0	Stores whole numbers from -128 to 127
short	2 bytes/ 16bits	0	Stores whole numbers from -32,768 to 32,767
int	4 bytes/ 32bits	0	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes/ 64bits	0L	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes/ 32bits	0.0f	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes/ 64bits	0.0d	Stores fractional numbers. Sufficient for storing 15 decimal digits
char	2 bytes/ 16bits	'\u0000'	Stores a single character/letter or ASCII values



# Atribuição de valor com outra variável

- O valor de uma variável pode ser utilizado para atribuir o valor de outra variável;
- Ou seja, iniciamos uma segunda variável com o valor de uma primeira, isso é aceito em Java;
- Outra consideração importante são os limites dos tipos de dados, que foram passados na tabela da aula passada;
- Vamos explorar estes dois cenários!





# Comentário em Java

- Comentários são usados para explicar o código, tornando-o mais fácil de entender para outros desenvolvedores;
- Comentários não afetam a execução do programa, são completamente ignorados pelo compilador;
- Tipos de comentários:
  - **Comentário de linha:** Usado para breves explicações em uma linha (`// Comentário`).
  - **Comentário de bloco:** Explicações mais longas em várias linhas (`/* Comentário */`).
  - **Comentário de documentação:** Usado para gerar documentação do código (`/** Comentário */`).
- Comentários ajudam a manter o código legível e compreensível, facilitando a manutenção;



# Strings

- Strings são sequências de caracteres usadas para armazenar e manipular texto;
- Classe String: Em Java, as Strings são objetos da classe String e oferecem métodos para manipulação de texto;
- Uma String não pode ser alterada, qualquer modificação gera uma nova String;
- Podemos concatenar (unir) duas strings com o operador +;
- Vamos testar!





# Char

- char é um tipo de dado primitivo que armazena um único caractere;
- O char pode armazenar qualquer caractere Unicode, incluindo letras, números e símbolos;
- Ocupa 2 bytes de memória, permitindo armazenar 65.536 caracteres diferentes;
- Vamos testar!



# Int

- int é um tipo de dado primitivo utilizado para armazenar números inteiros;
- Tamanho fixo: Ocupa 4 bytes de memória, permitindo armazenar valores de -2.147.483.648 a 2.147.483.647;
- Ideal para operações aritméticas e contadores em loops;
- Vamos testar!





# Long

- long é um tipo de dado primitivo usado para armazenar números inteiros muito grandes;
- Ocupa 8 bytes de memória, permitindo armazenar valores de -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807;
- Ideal para cálculos financeiros, contagem de tempo, ou quando o tipo int não é suficiente;
- Valores literais do tipo long devem ser seguidos por L (maiúsculo) para indicar o tipo;
- O sublinhado pode ser usado para melhorar a legibilidade em números longos, separando grupos de dígitos (Ex: 123\_456\_789\_012\_345L);
- Vamos testar!



# Double

- double é um tipo de dado primitivo usado para armazenar números de ponto flutuante (decimais) com dupla precisão;
- Ocupa 8 bytes de memória, permitindo armazenar uma ampla gama de valores decimais;
- Ideal para operações matemáticas que requerem precisão, como cálculos científicos e financeiros;
- Valores literais podem ser seguidos por d ou D, embora não seja obrigatório;
- O sublinhado pode ser usado para separar grupos de dígitos em números longos para melhorar a legibilidade;
- Vamos testar!





# Operadores aritméticos parte 1

- Adição (+): Soma de dois valores;
- Subtração (-): Subtração de um valor pelo outro;
- Multiplicação (\*): Multiplicação de dois valores;
- Divisão (/): Divisão de um valor pelo outro;
- Módulo (%): Resto da divisão de dois valores;
- Vamos testar!



# Operadores aritméticos parte 2

- Incremento (++): Incrementa o valor de uma variável em 1;
- Decremento (--): Decrementa o valor de uma variável em 1;
- Atribuição aditiva (+=): Soma e atribui o resultado a uma variável;
- Atribuição subtrativa (-=): Subtrai e atribui o resultado a uma variável;
- Vamos testar!





# Type casting

- Casting implícito (widening): Converte automaticamente tipos menores para tipos maiores (por exemplo, int para long), sem perda de dados;
- Casting explícito (narrowing): Necessário quando se converte tipos maiores para tipos menores (por exemplo, double para int), podendo resultar em perda de dados;
- Para realizar um casting explícito, é necessário especificar o tipo de destino entre parênteses;
- Vamos testar!



# Exercícios

- Vamos agora fazer os exercícios da seção de Variáveis!





# Constantes em Java com final

- final: Define uma variável como constante, impedindo que seu valor seja alterado após a inicialização;
- Uma vez atribuído, o valor não pode ser modificado;
- Boas práticas: Usado para valores que não devem mudar durante a execução do programa, como PI ou taxas de juros;
- Convenção de nomes: Constantes geralmente são nomeadas em letras maiúsculas, com palavras separadas por sublinhados (\_);



# Inferência de Tipo com var em Java

- var: Introduzido no Java 10, permite ao compilador inferir o tipo da variável com base no valor atribuído;
- Reduz a necessidade de escrever tipos longos e complexos, aumentando a legibilidade do código;
- Tipo estático: Embora o tipo seja inferido, ele é fixo após a atribuição e não pode ser alterado;
- Regras: Deve ser inicializado no momento da declaração, e não pode ser usado para variáveis não inicializadas;
- Boas práticas: Útil para tipos complexos ou quando o tipo é óbvio a partir do contexto;





# A classe Scanner

- A classe Scanner é utilizada para ler a entrada de dados do usuário via console/terminal;
- Parte do pacote java.util: Para usar o Scanner, é necessário importar a classe do pacote java.util;
- Scanner pode ler diferentes tipos de dados, como int, double, String, etc;
- Métodos comuns:
  - `nextLine()`: Lê uma linha inteira de texto;
  - `nextInt()`: Lê um valor inteiro;
  - `nextDouble()`: Lê um valor decimal (ponto flutuante);
  - `next()` : Lê uma única palavra;



# Importação de pacotes

- Pacotes organizam classes e interfaces relacionadas, melhorando a estrutura do código;
- Importar classes de pacotes externos, como Scanner, é necessário para utilizá-las;
- A importação de pacotes previne conflitos de nomes entre classes em projetos grandes;
- Permite a reutilização de código existente, acelerando o desenvolvimento;
- Facilita a manutenção e a leitura do código ao manter uma organização lógica;





# Fechamento do scanner

- O Scanner consome recursos de entrada, como fluxo de dados do teclado;
- Fechar o Scanner libera esses recursos, evitando problemas de performance;
- Deixar o Scanner aberto pode causar vazamentos de memória ou travamentos;
- `close()` é uma boa prática recomendada após a leitura dos dados;
- O fechamento do Scanner ajuda a manter a aplicação eficiente e estável;



# Problema do nextLine

- `nextLine()` lê a linha inteira até encontrar um Enter;
- Problema ocorre ao usar `nextLine()` após `nextInt()`, `nextDouble()`, etc;
- `nextLine()` captura o caractere Enter remanescente, resultando em uma leitura vazia;
- Isso faz o programa parecer "pular" a entrada de texto;
- Solução: adicionar um `nextLine()` extra após a leitura de números;
- Vamos ver na prática;







# Variáveis

Conclusão



# Condicionalis

Introdução



# O que é boolean?

- boolean é um tipo de dado primitivo em Java que pode armazenar apenas dois valores: true ou false;
- Utilizado para controlar o fluxo de execução em estruturas de controle como if, while, e for;
- Suporta operadores lógicos como && (AND), || (OR), e ! (NOT) para combinar e inverter valores booleanos;
- Uma expressão booleana é qualquer expressão que resulta em um valor true ou false;
- Embora seja representado como true ou false no código, internamente é tratado como 1 (true) ou 0 (false);



# Operadores de comparação

- == (Igual a): Verifica se dois valores são iguais;
- != (Diferente de): Verifica se dois valores são diferentes;
- > (Maior que): Verifica se o valor à esquerda é maior que o valor à direita;
- < (Menor que): Verifica se o valor à esquerda é menor que o valor à direita;
- >= (Maior ou igual a): Verifica se o valor à esquerda é maior ou igual ao valor à direita;
- <= (Menor ou igual a): Verifica se o valor à esquerda é menor ou igual ao valor à direita;





# Diferença entre comparação e atribuição

- Atribuição (=):
  - Atribui um valor a uma variável;
  - Usado para definir ou alterar o valor armazenado em uma variável;
  - Exemplo: `int a = 5;`
- Comparação (==):
  - Compara dois valores para verificar se são iguais;
  - Retorna true se os valores forem iguais e false se forem diferentes;
  - Exemplo: `5 == 5` retorna true;



# Comparação de strings

- Problema com ==:
  - O operador == compara as referências de memória, não o conteúdo das strings;
  - Pode retornar false mesmo que o conteúdo das strings seja igual, se as referências forem diferentes;
- Uso do método equals():
  - equals() compara o conteúdo das strings, caractere por caractere;
  - É a maneira correta e segura de verificar se duas strings são iguais em valor;
- equalsIgnoreCase():
  - Variante de equals() que ignora diferenças entre maiúsculas e minúsculas;





# Estruturas de condição

- if: Executa um bloco de código se a condição for verdadeira;
- else: Executa um bloco de código alternativo se a condição do if for falsa;
- else if: Verifica outra condição se as condições anteriores forem falsas;
- switch: Seleciona e executa um bloco de código entre várias opções com base no valor de uma expressão;
- Importante: Todas essas estruturas permitem controlar o fluxo de execução com base em condições lógicas;



# Conhecendo o if

- if executa um bloco de código se a condição for verdadeira;
- Coloque a condição entre parênteses após a palavra-chave if;
- Uso comum: Comparações lógicas ou aritméticas para tomar decisões no código;
- Importante: O bloco de código dentro de if é delimitado por chaves {};





# Explorando o else

- else executa um bloco de código quando a condição do if é falsa;
- Proporciona uma alternativa no fluxo de execução do programa;
- Sintaxe: O bloco else vem logo após um bloco if;
- Bloco único: Somente um bloco else pode seguir um if;
- Importante: Sempre use {} para delimitar o bloco de código do else;



# Utilizando o else if

- else if permite testar condições adicionais após um if;
- Sintaxe: Coloque a condição entre parênteses após a palavra-chave else if;
- Uso comum: Quando há várias condições mutuamente exclusivas;
- Encadeamento: Vários blocos else if podem ser usados após um if;
- Bloco final opcional: Pode ser seguido de um else para lidar com qualquer caso não coberto;





# Operadores lógicos

- && (E lógico): Retorna true se ambas as condições forem verdadeiras;
- || (OU lógico): Retorna true se pelo menos uma das condições for verdadeira;
- ! (NÃO lógico): Inverte o valor lógico; retorna true se a condição for falsa e vice-versa;
- Combinação: Pode combinar múltiplas condições em uma única expressão lógica;



# Tabela verdade

- A tabela verdade simula todas as combinações possíveis dos operadores lógicos, e exibe os resultados:

AND			OR			NOT	
A	B	$A \cup B$	A	B	$A \cup B$	A	$\bar{A}$
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		





# Trabalhando com o AND

- O operador && (AND lógico) retorna true se ambas as condições forem verdadeiras.
- Sintaxe: Condição1 && Condição2;
- Curto-circuito: Se a primeira condição for false, a segunda condição não é avaliada;
- Uso comum: Combinação de múltiplas condições que precisam ser verdadeiras ao mesmo tempo;



# Operador OR

- O operador || (OR lógico) retorna true se pelo menos uma das condições for verdadeira;
- Sintaxe: Condição1 || Condição2;
- Curto-circuito: Se a primeira condição for true, a segunda condição não é avaliada;
- Uso comum: Verificação de múltiplas condições onde apenas uma precisa ser verdadeira;





# Operador NOT

- O operador ! (NOT lógico) inverte o valor lógico de uma expressão;
- Sintaxe: ! seguido da condição ou expressão;
- Uso comum: Negar uma condição para tomar decisões baseadas no oposto;
- Útil em validações: Verificar se uma condição é falsa, ao invés de verdadeira;
- Combinação: Pode ser combinado com outros operadores lógicos (&&, ||) para criar expressões mais complexas;



# Estrutura switch em Java: case e break

- switch: Estrutura de controle que permite escolher entre várias opções com base no valor de uma expressão;
- case: Define uma possível opção ou caminho dentro do switch. Cada case é seguido por um valor específico que é comparado com a expressão do switch;
- break: Utilizado para encerrar a execução de um bloco case. Evita que o código "caia" nos casos seguintes;
- Importante: Cada case deve terminar com um break (ou outro comando de desvio) para evitar a execução de outros casos;
- Valores exclusivos: Os valores em case devem ser exclusivos e correspondentes ao tipo da expressão do switch;





# Estrutura switch em Java: default

- default: O bloco default é executado quando nenhum dos valores especificados nos case corresponde à expressão do switch;
- Opcional: O uso de default é opcional, mas recomendado para capturar todos os casos não previstos;
- Posição: Normalmente é colocado no final do switch, mas pode aparecer em qualquer lugar dentro do bloco;
- Sem break necessário: Como default geralmente é o último bloco, não é necessário usar break, mas pode ser incluído se o default não for o último;
- Fornece um comportamento padrão ou uma mensagem de erro quando nenhum caso específico é atendido;



# Switch sem brake

- fall-through: Sem break, o switch continua executando os blocos subsequentes, mesmo que o caso correspondente já tenha sido encontrado;
- Efeito inesperado: Pode levar à execução de múltiplos casos, causando resultados inesperados;
- Necessário break: Para interromper a execução após o bloco case correspondente, evitando o efeito de fall-through;
- Uso intencional: Em raros casos, o fall-through pode ser usado intencionalmente, mas é menos comum e deve ser bem documentado;
- Boa prática: Sempre incluir break para prevenir comportamentos indesejados, a menos que o fall-through seja intencional;





# Quando Usar if vs. switch

- Usar if:
  - Ideal para expressões booleanas simples;
  - Perfeito para condições que envolvem operadores lógicos (&&, ||) e comparações entre diferentes variáveis;
  - Útil quando a condição depende de um intervalo de valores (ex.:  $x > 10$ );
  - Útil para comparar objetos, strings com equals(), ou outras condições não numéricas;
- Usar switch:
  - Melhor para escolher entre várias opções discretas baseadas em um único valor;
  - Ideal quando você está lidando com um conjunto limitado de valores, como enumerações ou dias da semana;
  - Facilita a leitura e a manutenção quando há muitas opções (mais de 3 ou 4 casos);
  - Funciona bem com expressões baseadas em inteiros, caracteres, strings ou enums;



# Exercícios

- Vamos agora fazer os exercícios da seção de Condicionais!





# Condicionais ternárias

- Uma forma compacta de expressar uma condição if-else;
- Sintaxe: condição ? expressão1 : expressão2;
- Funcionamento: Avalia a condição; se for verdadeira, retorna expressão1, caso contrário, retorna expressão2;
- Ideal para atribuições simples e condições em linha;
- Limitação: Deve ser usado apenas em expressões simples para manter a legibilidade;



# If e else aninhado

- Estrutura onde um if ou else contém outro bloco if-else;
- Uso comum: Para testar múltiplas condições que dependem umas das outras;
- Útil para lidar com decisões complexas, mas pode prejudicar a legibilidade se usado em excesso;
- Sintaxe: Blocos if e else podem conter outros if-else, criando um encadeamento de condições;
- Boa prática: Mantenha o código claro e evite encadeamentos profundos;





# Precedência de Operadores Lógicos

- A ordem em que os operadores lógicos e de comparação são avaliados em uma expressão;
- Ordem de Precedência:
  - 1. (): Parênteses têm a maior precedência e são avaliados primeiro;
  - 2. !: Operador NOT lógico tem a segunda maior precedência;
  - 3. &&: Operador AND lógico é avaliado antes de ||;
  - 4. ||: Operador OR lógico é avaliado por último;
- Compreender a precedência evita erros lógicos e garante que as expressões sejam avaliadas conforme esperado;
- Uso de Parênteses: Parênteses podem ser usados para alterar a ordem de avaliação e melhorar a clareza do código;





# Condicionalis

Conclusão





# Funções

Introdução

# O que é uma função?

- Um **bloco de código** que realiza uma tarefa específica e pode ser chamado para ser executada;
- **Divide o código em partes menores**, tornando-o mais organizado e fácil de manter;
- Permite reutilizar código em diferentes partes do programa **sem precisar reescrever** as mesmas instruções;
- **Parâmetros e Retorno**: Pode receber dados de entrada (parâmetros) e retornar um resultado após a execução;
- As variáveis declaradas dentro de uma função **são locais** e não afetam o restante do programa;





# Criando a primeira função

- **Normalmente** uma função em Java é definida com um tipo de retorno, um nome e pode ou não receber parâmetros;
- A função **pode ser criada sem parâmetros e sem retorno**, ideal para tarefas simples que não requerem entrada ou saída;
- **Sintaxe Básica:** Consiste em um cabeçalho que inclui o tipo de retorno (void para sem retorno) e o corpo da função, onde o código é executado;
- **Chamando a Função:** A função é invocada pelo seu nome, e o código dentro dela é executado sempre que chamada;



# Diferença entre a Função main e Outras Funções

- **Função main:**

- É o ponto de entrada do programa, onde a execução começa;
- Deve ter a assinatura exata `public static void main(String[] args);`
- Todo programa Java precisa de uma função main para ser executado;
- A função main pode chamar outras funções e métodos dentro do programa;

- **Outras Funções:**

- Criadas para dividir o código em partes menores e gerenciáveis;
- Podem ter diferentes tipos de retorno, nomes, e receber parâmetros.;
- Podem ser chamadas várias vezes em diferentes partes do programa;
- São executadas apenas quando chamadas pelo código, ao contrário do main, que é executado automaticamente;





# Argumentos em funções

- **Dados que você passa para uma função ao chamá-la**, permitindo que a função processe informações específicas;
- Parâmetros são **variáveis definidas na assinatura da função** para receber os argumentos;
- Funções podem receber nenhum, um ou vários argumentos, dependendo da tarefa que realizam;
- **Argumentos tornam as funções mais flexíveis** e reutilizáveis em diferentes contextos com dados diferentes;
- **Tipos de Argumentos:** Podem ser de qualquer tipo primitivo (int, double, etc.) ou objetos;



# Uso do return em funções

- **Uma instrução que finaliza a execução de uma função** e, opcionalmente, devolve um valor ao ponto onde a função foi chamada;
- **Finalização de Função:** Quando o return é executado, a função para de executar, e o controle é devolvido ao chamador;
- **Tipos de Retorno:** O return pode retornar valores de qualquer tipo, incluindo tipos primitivos, objetos, ou nenhum valor (void);
- O return permite que funções realizem cálculos ou operações e **enviem o resultado de volta** para ser utilizado em outras partes do programa;





# Funções com X sem retorno

- **Funções com Retorno:**

- Permitem que uma operação seja realizada e seu resultado seja utilizado em outras partes do programa;
- Cálculos, validações, e operações que produzem um resultado necessário para outras funções ou partes do código;
- Exemplo: Calcular a soma de dois números e retornar o resultado para ser exibido ou usado em outro cálculo;

- **Funções sem Retorno:**

- Executam uma ação sem precisar devolver um resultado, ideal para tarefas como exibição de dados ou alterações diretas no estado do programa;
- Exibir mensagens, modificar variáveis globais, ou realizar operações que não requerem um retorno;
- Exemplo: Exibir uma mensagem de boas-vindas ou atualizar o valor de uma variável de controle;



# Encapsulando retorno em variável

- O processo de **armazenar o resultado de uma função em uma variável** para uso posterior;
- Permite reutilizar o valor retornado por uma função em **várias partes do código**, aumentando a modularidade e a legibilidade;
- **Uso Comum:** Armazenar resultados de cálculos, verificações ou operações complexas para evitar chamadas repetidas à mesma função;
- Encapsular o retorno em uma variável **pode ajudar a simplificar o código** e reduzir a necessidade de executar novamente a função;





# Funções com if/else e Condicionais Complexas

- Estruturas de controle if/else dentro de funções **permitem tomar decisões complexas** baseadas em múltiplas condições;
- Condicionais Complexas: Permitem **combinar múltiplas condições** usando operadores lógicos (&&, ||) e comparações para determinar o fluxo de execução;
- **Uso Comum:** Verificações de múltiplos critérios, tomadas de decisão em processos complexos e validações de entradas de usuário;
- Funções com if/else **organizam e centralizam a lógica de decisão**, tornando o código mais modular e fácil de manter;



# Funções com switch

- O switch é uma estrutura de controle que permite a execução de diferentes blocos de código com base no valor de uma expressão;
- O switch dentro de uma função é útil **quando há múltiplas condições discretas (casos)** a serem verificadas, como valores inteiros, caracteres ou strings;
- **Benefícios:** Simplifica a lógica quando comparado a múltiplos if-else, tornando o código mais organizado e fácil de entender;
- **Boa Prática:** Sempre incluir um default para tratar valores inesperados ou casos não cobertos;





# Funções com System.exit

- **O que é System.exit:**
  - O método `System.exit(int status)` encerra imediatamente a execução do programa, finalizando todas as operações em andamento;
  - O argumento `int status` indica o estado de término do programa, um valor de 0 geralmente indica uma saída bem-sucedida, enquanto valores diferentes de 0 indicam erros;
  - Como o `System.exit` termina o programa abruptamente, deve ser usado com cuidado, normalmente em situações de erro crítico ou quando não há mais nada a ser feito;
- **Casos de Uso:**
  - Tratamento de Erro: Em cenários onde o programa não pode continuar devido a um erro crítico;
  - Interrupção Controlada: Quando o programa atinge um estado em que deve ser encerrado imediatamente, como após confirmar a saída do usuário;



# Documentando funções

- **O que é Documentação de Função:**

- Fornece informações detalhadas sobre o que a função faz, seus parâmetros, valor de retorno, e outros detalhes relevantes;
- Utiliza o formato Javadoc (`/** ... */`) para gerar documentação automática e legível, que pode ser visualizada em IDEs e ferramentas de documentação;
- Facilita a compreensão e manutenção do código, especialmente em projetos colaborativos ou de grande escala;

- **Componentes da Documentação:**

- Descrição Geral: Explica o propósito da função e o que ela faz;
- Parâmetros (`@param`): Descreve os parâmetros de entrada, incluindo seus tipos e o que representam;
- Valor de Retorno (`@return`): Descreve o que a função retorna, se aplicável;
- Exceções (`@throws`): Indica quais exceções a função pode lançar, se houver;





# Escopos em Java

- Escopo refere-se à **visibilidade** e ao **tempo de vida de variáveis** e objetos dentro de um programa;
- **Escopo Local:** Variáveis declaradas dentro de métodos, blocos if, for, ou while só são acessíveis dentro desses blocos;
- **Escopo de Classe (Global):** Variáveis declaradas fora de métodos, mas dentro de uma classe, são acessíveis por todos os métodos dessa classe;
- **Escopo de Parâmetro:** Parâmetros de função são tratados como variáveis locais dentro do escopo da função;
- **Encapsulamento:** O escopo ajuda a proteger variáveis e métodos de acessos indesejados fora do seu contexto apropriado;



# O que são Funções Built-in em Java?

- São funções já incorporadas na linguagem Java que **fornecem funcionalidades comuns e essenciais**. Elas são prontas para uso e não precisam ser definidas pelo programador;
- Estão **disponíveis automaticamente** sem necessidade de importação ou definição;
- Para que servem: manipulação de strings, operações matemáticas, conversão de dados, entre outras funcionalidades;
- Geralmente, as funções built-in são **altamente otimizadas** para desempenho;





# Funções Built-in de String

- **length():** Retorna o comprimento de uma string, ou seja, o número de caracteres;
- **substring(int beginIndex, int endIndex):** Extrai uma subsequência da string, começando no índice beginIndex e terminando em endIndex;
- **toUpperCase():** Converte todos os caracteres da string para letras maiúsculas;
- **replace(char oldChar, char newChar):** Substitui todas as ocorrências de um caractere especificado por outro;
- Vamos ver na prática!



# Funções Built-in de Números (Math)

- **Math.sqrt(double a):** Calcula a raiz quadrada de um número;
- **Math.pow(double a, double b):** Eleva um número a à potência b;
- **Math.abs(int a):** Retorna o valor absoluto de um número;
- **Math.max(int a, int b):** Retorna o maior de dois números;
- Vamos utilizá-las!





# Exercícios

- Vamos agora fazer os exercícios da seção de Funções!



# Funções recursivas

- Recursão é a técnica onde **uma função chama a si mesma** para resolver um problema que pode ser dividido em subproblemas menores e semelhantes ao original;
- **Um caso base/cenário é essencial para terminar a recursão**, sem ele, a função entraria em um loop infinito;
- **Como Funciona:** O problema é dividido em subproblemas menores até atingir o caso base, após o qual a solução começa a ser construída ao "subir" a pilha de chamadas;
- **Pode ser ineficiente em termos de tempo e memória**, especialmente para problemas grandes, devido à sobrecarga de chamadas de função;





# Sobrecarga de Funções (Method Overloading)

- Sobrecarga de funções permite definir várias funções com o mesmo nome, desde que tenham assinaturas diferentes (número ou tipo de parâmetros);
- **Vantagens:**
  - Permite criar diferentes versões de uma função para lidar com diferentes tipos de dados ou diferentes quantidades de argumentos;
  - Mantém o código limpo e organizado, reutilizando o nome da função para tarefas relacionadas;
- **Regras para Sobrecarga:**
  - Número de Parâmetros: As funções devem diferir no número de parâmetros;
  - Tipo de Parâmetros: As funções podem ter o mesmo número de parâmetros, desde que os tipos sejam diferentes;
  - Tipo de Retorno: Não pode ser usado sozinho para diferenciar funções sobrecarregadas;



# Funções Anônimas (Lambda Expressions)

- **Lambdas são funções anônimas**, ou seja, funções sem nome, que podem ser usadas para expressar brevemente pequenas operações ou blocos de código, especialmente em programação funcional;
- **Introduzidas no Java 8**, as lambdas são uma parte central da API Streams e permitem uma programação mais funcional;
- **Sintaxe:** Formato: (parâmetros) -> { corpo da função }
- Para expressões simples, o corpo da função pode ser uma única linha sem { };
- **Vantagens:**
  - Lambdas permitem escrever código mais conciso e legível;
  - São amplamente utilizadas em conjunto com Streams e interfaces funcionais como Runnable e Comparator;







# Funções

Conclusão



# Estruturas de repetição

Introdução



# O que são estruturas de repetição?

- Estruturas de repetição, ou loops, são comandos que permitem a execução repetida de um bloco de código enquanto uma condição específica for verdadeira;
- Para que servem:
  - Automatizar a execução de tarefas repetitivas, economizando tempo e esforço ao evitar a necessidade de escrever o mesmo código várias vezes;
  - Permitem iterar sobre coleções de dados, realizar operações com números sequenciais, e processar entradas de forma eficiente;
- As mais utilizadas:
  - for: Ideal para quando se sabe o número exato de iterações a serem realizadas. Permite o controle de variáveis de início, condição de continuidade e incremento;
  - while: Executa um bloco de código enquanto uma condição é verdadeira. Útil quando o número de iterações não é conhecido antecipadamente;
  - do-while: Similar ao while, mas garante que o bloco de código seja executado pelo menos uma vez, pois a condição é verificada após a execução;



# Estrutura for

- **for** é uma estrutura de repetição que permite executar um bloco de código um número determinado de vezes;
- Ideal para quando se sabe o **número exato de iterações**;
- **Consiste em três partes:** inicialização, condição e incremento;
- Oferece controle sobre o fluxo de repetição com variáveis de início e incremento;





# Estrutura de repetição while

- **while** é uma estrutura de repetição que executa um bloco de código repetidamente enquanto uma condição específica for verdadeira;
- **Sintaxe:** while (condição) { bloco de código }
- Execução contínua até que uma condição seja falsa, ideal para **quando o número de iterações não é conhecido** antecipadamente;



# Loop infinito

- Um loop infinito é uma **estrutura de repetição que nunca termina**, continuando a executar indefinidamente porque a condição de término nunca é atendida;
- **Causas Comuns:** Condição de saída mal definida, ausência de incremento ou decremento de variáveis de controle;
- **Cuidado:** Pode causar o travamento do programa ou consumo excessivo de recursos, exigindo atenção na implementação;





# Do while

- **do-while** é uma estrutura de repetição que garante que o bloco de código seja executado pelo menos uma vez, verificando a condição após a execução;
- **Sintaxe:** `do { bloco de código } while (condição);`
- **Ideal quando é necessário executar o código ao menos uma vez** antes de verificar a condição;



# Break em loops

- **break** é uma instrução usada para interromper um loop imediatamente, mesmo que a condição de término original não tenha sido atingida;
- Frequentemente **usado para sair de loops** antecipadamente quando uma condição específica é satisfeita;
- Termina o loop atual e continua a execução do código após o loop;





# Continue em loops

- **continue** é uma instrução que interrompe a iteração atual do loop e pula para a próxima, ignorando o restante do código dentro do bloco do loop para aquela iteração;
- Usado para **pular certas iterações** quando uma condição específica é atendida, sem interromper o loop completo;
- **Continua a execução do loop na próxima iteração**, ignorando as instruções após o continue na iteração atual;



# Nested loops

- Nested loops ocorrem quando **um loop é colocado dentro de outro loop**, permitindo que o loop interno seja executado completamente para cada iteração do loop externo;
- Frequentemente usados para **manipulação de matrizes**, tabelas e para iterar sobre estruturas de dados mais complexas;
- Podem ser menos eficientes e mais **complexos de entender**, exigindo cuidado para evitar loops infinitos ou comportamento inesperado;





# Exercícios

- Vamos agora fazer os exercícios da seção de Loops!



# Loops com rótulos

- Rótulos (**labels**) em Java são usados para identificar blocos específicos de código, como loops. Eles permitem o controle direto sobre qual loop deve ser interrompido ou continuado, especialmente em loops aninhados;
- **Rótulos são utilizados em conjunto com break e continue** para sair ou pular diretamente para loops externos, em vez de apenas o loop interno;





# Erro Comum: Loops Off-by-One

- O erro "off-by-one" ocorre quando um **loop executa uma iteração a mais ou a menos** do que o necessário devido a um erro na configuração da condição de término;
- Causa: Erro na condição de comparação, **como usar  $\leq$  em vez de  $<$** , ou começar o loop no índice errado;
- **Pode levar a resultados inesperados**, como omissão de dados ou acesso a índices inválidos em arrays;





# Estruturas de repetição

Conclusão





# Arrays

Introdução

# O que são arrays?

- Arrays são **estruturas de dados** que armazenam múltiplos valores do mesmo tipo em uma única variável;
- Arrays **podem armazenar tipos primitivos** (como int, double, char) ou objetos (como String ou classes personalizadas);
- Os elementos de um array são acessados por índices, **que começam em 0**;
- O tamanho de um array é definido na sua criação e não pode ser alterado;
- **Arrays são usados para armazenar listas de dados**, como números, nomes, ou qualquer coleção de elementos homogêneos;





# Sintaxe de arrays

- Arrays são declarados **especificando o tipo** dos elementos seguido de colchetes ([]);
- Arrays podem ser inicializados com valores específicos no momento da declaração ou instanciados com um **tamanho fixo**;
- Os elementos de um array são acessados usando **índices que começam em 0**;
- **Sintaxe:** `tipo[] nomeArray = new tipo[tamanho];` ou `tipo[] nomeArray = {valores};`



# Loop em arrays

- Loops são usados para **iterar sobre os elementos** de um array, acessando e manipulando cada elemento individualmente;
- Você pode usar **for**, **while**, ou loops aprimorados (for-each) para iterar sobre arrays;
- Percorrer todos os elementos, somar valores, encontrar elementos específicos, modificar valores;





# Loop for-each em Arrays

- O loop for-each permite iterar sobre os elementos de um array de maneira simplificada, **sem necessidade de gerenciar índices**;
- Ideal para percorrer todos os elementos de um array quando não é necessário modificar o índice ou acessar elementos fora da sequência;
- Sintaxe: `for (tipo variavel : array) { // bloco de código }`



# For x For-each

- **Usar for:**

- Ideal quando você precisa acessar ou manipular o índice de cada elemento, como em casos onde o índice determina a lógica ou o comportamento do código;
- Útil quando a iteração precisa pular elementos, ou quando o loop deve ser interrompido ou reiniciado em uma posição específica;
- Recomendado quando você precisa modificar diretamente os elementos do array usando o índice;

- **Usar for-each:**

- Ideal para percorrer todos os elementos de um array ou coleção sem a necessidade de modificar os elementos ou controlar o índice;
- Melhor para código que precisa ser limpo e fácil de entender, onde a manipulação do índice não é necessária;
- Evita erros relacionados ao índice, como `ArrayIndexOutOfBoundsException`, porque o for-each não expõe o índice diretamente;





# Loops com if

- Combinar loops com instruções if permite **executar condições específicas dentro de cada iteração** do loop;
- Usos: Filtrar elementos, interromper ou continuar a execução do loop com base em condições lógicas;
- Usar if dentro de loops adiciona lógica condicional, permitindo decisões dinâmicas em cada iteração;



# Atualizações de valores em arrays

- Atualizar valores em um array significa **modificar os elementos do array** em índices específicos, seja através de loops ou diretamente;
- Usos: Alterar valores com base em cálculos, substituir elementos, ou aplicar operações em todos os elementos do array;
- Permite manipular dados armazenados em arrays para **refletir mudanças dinâmicas** ou realizar operações em massa;





# Método toString de Arrays

- O método toString da classe Arrays é usado para **converter um array em uma representação em String**, permitindo a exibição direta do conteúdo do array;
- Uso Comum: **Facilita a visualização dos elementos** de um array em uma única linha, útil para debug e saída de dados;
- Sintaxe: `Arrays.toString(array)` converte o array em uma string formatada;



# Maneiras de Adicionar Novos Itens a Arrays

- Arrays em Java têm tamanho fixo, portanto, para adicionar novos itens, é necessário criar um novo array ou usar estruturas como ArrayList;
- Métodos Comuns: Criar um novo array maior e copiar os elementos, usar ArrayList para manipulação dinâmica de elementos;





# Reference Trap

- Reference trap ocorre quando **duas variáveis apontam para o mesmo objeto** na memória, causando modificações não intencionais ao objeto original ao alterar a cópia;
- Ao copiar arrays ou objetos, **a nova variável pode compartilhar a referência ao mesmo espaço de memória**, em vez de criar uma nova instância;
- Alterações em uma variável refletem na outra, pois ambas referenciam o mesmo objeto;



# Arrays 2D (matrizes)

- Arrays 2D são **arrays de arrays**, onde cada elemento do array principal é um outro array, permitindo a criação de estruturas de grade (**matriz**);
- Um array 2D é criado **especificando o número de linhas e colunas**;
- Os elementos são acessados usando dois índices: um para a linha e outro para a coluna;
- Elementos de um array 2D podem ser atribuídos diretamente ou através de loops;





# Exercícios

- Vamos agora fazer os exercícios da seção de Arrays!



# Ordenação de arrays

- **Ordenação com `Arrays.sort()`:**
  - Ordena arrays de tipos primitivos em ordem crescente;
  - Ordena arrays de objetos que implementam `Comparable`;
  - Pode usar `Comparator` para definir ordem personalizada;
- **Ordenação de Arrays Multidimensionais:**
  - Ordenação baseada em uma coluna específica;
  - `Comparator` pode ser utilizado para definir critérios complexos;





# Manipulação avançada de arrays

- Cópia de Arrays com **Arrays.copyOf()**:
  - Cria uma nova cópia de um array com tamanho especificado;
  - Pode ser utilizado para truncar ou expandir arrays;
- Preenchimento de Arrays com **Arrays.fill()**:
  - Preenche todos os elementos do array com um valor específico;
  - Útil para inicialização de arrays com valores padrão;
- Transformação de Arrays com **Arrays.stream()**:
  - Converte um array em um Stream para manipulação funcional;
  - Permite filtragem, mapeamento, redução e outras operações;



# Array dinâmico

- Conceito de Arrays Dinâmicos:
  - Arrays em Java são de tamanho fixo; arrays dinâmicos **ajustam o tamanho automaticamente**;
  - **ArrayList** é a implementação mais comum de arrays dinâmicos;
- Adicionar Elementos com **ArrayList.add()**:
  - Permite adicionar elementos ao final do array;
  - Não requer redimensionamento manual;
- Remover Elementos com **ArrayList.remove()**:
  - Remove elementos pelo índice ou valor;
  - Ajusta automaticamente o tamanho do array;







# Arrays

Conclusão



# Orientação a Objetos

Introdução



# O que é Orientação a Objetos?

- Orientação a Objetos (OOP) é um **paradigma de programação** que organiza o software em torno de objetos;
- Esses objetos **representam entidades do mundo real** e interagem entre si para resolver problemas;
- **Cada objeto é uma instância de uma classe**, que define os atributos e comportamentos do objeto;
- OOP facilita o **design modular** do software, tornando-o mais fácil de manter, reutilizar e expandir;
- Os principais pilares da OOP são **abstração, encapsulamento, herança e polimorfismo**;
- Benefícios da Orientação a Objetos
  - **Modularidade:** O código é organizado em classes e objetos, tornando-o mais fácil de entender e manter;
  - **Reuso de Código:** A herança e a modularidade facilitam a reutilização de partes do código em diferentes aplicações;
  - **Facilidade de Manutenção:** Alterações podem ser feitas em uma classe sem afetar outras partes do sistema, promovendo maior flexibilidade;
  - **Segurança:** O encapsulamento protege os dados e controla o acesso aos atributos e métodos de uma classe;



# Terminologia de OO

- **Classe**
  - Um "molde" ou "blueprint" para criar objetos;
  - Define os atributos (propriedades) e métodos (comportamentos) de um objeto;
- **Objeto**
  - Instância de uma classe;
  - Exemplo: Um carro é um objeto da classe "Carro", com atributos como cor, modelo e métodos como acelerar, frear;
- **Atributo (ou Propriedade)**
  - São variáveis dentro da classe que armazenam os dados do objeto;
  - Exemplo: No objeto "Carro", atributos podem ser cor, modelo, ano;
- **Método**
  - Função definida dentro de uma classe que representa uma ação ou comportamento do objeto;
  - Exemplo: No objeto "Carro", métodos podem ser acelerar(), frear();
- **Instanciação**
  - Processo de criar um novo objeto a partir de uma classe;
  - Utiliza-se a palavra-chave new em Java para instanciar um objeto;





# Criando uma classe

- Uma classe é um **modelo** que define as **propriedades** e **comportamentos** de um objeto;
- Classes são criadas usando a **palavra-chave** **class**;
- O nome da classe deve ser um substantivo e **começar com letra maiúscula**;
- Dentro de uma classe, você pode definir **atributos (propriedades)** e **métodos (comportamentos)**;
- Uma classe não faz nada por si só até que seja **instanciada** (criação de um objeto);



# Instanciando objetos

- Objetos são **instâncias de uma classe**;
- Para instanciar um objeto, usamos a **palavra-chave new**;
- A instância **cria uma cópia do modelo** da classe em memória;
- Cada objeto **tem seus próprios valores de atributos**, independentes de outros objetos;
- A instância de um objeto **permite acessar métodos e atributos** definidos na classe;
- A sintaxe básica para criar um objeto é:
  - `NomeDaClasse nomeDoObjeto = new NomeDaClasse();`
- Ao instanciar um objeto, podemos acessar seus métodos e definir valores para seus atributos;





# Criando métodos

- Métodos **são funções dentro de uma classe** que definem o comportamento de um objeto;
- Eles permitem que objetos realizem **ações ou operações**;
- A sintaxe de um método inclui o **tipo de retorno, o nome do método, e os parâmetros**;
- **Métodos podem ou não retornar um valor**, dependendo da definição;
- Para criar um método, usamos a seguinte estrutura:

```
public TipoDeRetorno nomeDoMetodo(TipoParametro parametro) {  
    // Corpo do método  
}
```

- Se o método não retornar valor, **usamos void**;
- Os métodos **são chamados a partir de uma instância de um objeto** da classe;
- Os métodos podem receber parâmetros para trabalhar com dados e retornar resultados;



# O que é Encapsulation?

- **Encapsulamento (encapsulation)** é um dos pilares da Programação Orientada a Objetos (OOP);
- Ele consiste em **esconder os detalhes internos de uma classe** e expor apenas o necessário para o usuário da classe;
- A ideia principal é **proteger os dados de acessos indevidos** ou modificações diretas;
- Encapsulamento é obtido através do uso de modificadores de acesso, como **private**, **protected**, e **public**;
- Propriedades privadas só podem ser acessadas ou modificadas por métodos específicos da classe (**getters e setters**);
- Getters e setters fornecem uma maneira controlada de **acessar e modificar os atributos da classe**;
- Encapsulamento promove a integridade dos dados e facilita a manutenção do código;
- O objetivo é garantir que os atributos da classe só sejam alterados de maneira controlada e válida;





# Criando propriedades

- Propriedades são atributos de uma classe que **definem as características de um objeto**;
- **Cada propriedade tem um tipo de dado**, como int, double, String, entre outros;
- As propriedades podem ser **públicas, privadas ou protegidas**, dependendo da visibilidade desejada;
- Propriedades **geralmente são declaradas no início da classe**;
- Propriedades públicas **podem ser acessadas diretamente**, enquanto propriedades privadas exigem métodos getters e setters;
- O valor das propriedades pode ser alterado diretamente (se público) ou por meio de métodos (se privado);
- Boas práticas de encapsulamento **sugerem o uso de propriedades privadas com métodos públicos** para acessar e modificar os valores;



# O this em objetos

- O **this** é uma **palavra-chave em Java usada dentro de métodos** de uma classe para se referir ao objeto atual;
- Ele é útil para **distinguir entre os atributos do objeto e os parâmetros do método** com o mesmo nome;
- O **this** é **frequentemente usado em setters e construtores** para referenciar os atributos do objeto;
- Sempre que um método de uma classe é chamado, **o this se refere à instância do objeto** que fez a chamada;
- O **this** também pode ser usado para encadear métodos (method chaining) ou chamar outros construtores;





# O que são Setters?

- Setters são métodos usados para **alterar os valores das propriedades** privadas de uma classe;
- Eles permitem o controle sobre **como os atributos de um objeto são modificados**;
- Setters seguem a convenção de nomeação: `setNomePropriedade()`;
- Setters **garantem que os valores dos atributos sejam válidos** antes de serem atribuídos;
- Com setters, é possível adicionar **lógica de validação ou manipulação** dos dados antes de atualizar o valor;



# O que são Getters?

- Getters são métodos usados para **acessar os valores das propriedades privadas** de uma classe;
- **Eles permitem que outras classes ou métodos leiam os atributos**, mas sem permitir modificá-los diretamente;
- Getters seguem a convenção de nomeação: `getNomePropriedade()`;
- Um getter **retorna o valor da propriedade correspondente** e não deve alterar o estado do objeto;
- É uma prática comum usar getters para manter o **princípio do encapsulamento**;
- Getters ajudam a garantir que os atributos de um objeto sejam acessados de forma controlada;





# Lógica no Getter e Setter

- Getters e Setters podem **conter lógica para validar, processar ou modificar os dados** antes de serem lidos ou gravados;
- A lógica nos Setters permite **validar os dados antes de armazená-los**, garantindo integridade;
- A lógica nos Getters pode **processar ou formatar os dados** antes de retorná-los;
- Isso melhora a segurança e o controle sobre como os atributos são manipulados;
- Alterar dados com lógica nos Getters e Setters **mantém o princípio do encapsulamento** e previne valores inválidos;



# Método dentro de método

- Em Java, **métodos podem chamar outros métodos** dentro da mesma classe;
- Essa prática ajuda a modularizar o código, **quebrando funcionalidades complexas em partes menores**;
- Chamar um método dentro de outro **promove a reutilização de código** e facilita a manutenção;
- Ao combinar métodos, **é possível realizar ações mais complexas** e evitar duplicação de código;
- Métodos dentro de métodos ajudam a simplificar a lógica, tornando o código mais claro e legível;





# O que são constructors?

- Construtores (constructors) são **métodos especiais usados para inicializar objetos**;
- Eles são **chamados automaticamente** quando um novo objeto é criado;
- O nome de um construtor deve ser o mesmo que o nome da classe;
- Construtores **não têm um tipo de retorno** (nem mesmo void);
- Podem ser usados para **atribuir valores iniciais aos atributos de um objeto**;
- Java fornece um construtor padrão se nenhum for definido, mas construtores personalizados permitem maior controle na criação do objeto;



# Exercícios

- Vamos agora fazer os exercícios da seção de POO!





# Modificadores de acesso

- Os modificadores de acesso **controlam a visibilidade dos membros de uma classe** (atributos e métodos);
- **public**: O membro pode ser acessado de qualquer lugar (dentro ou fora do pacote);
- **private**: O membro só pode ser acessado dentro da própria classe;
- **protected**: O membro pode ser acessado dentro da classe, suas subclasses e classes do mesmo pacote;
- O uso correto dos modificadores de acesso é fundamental **para aplicar o encapsulamento**;
- Eles protegem os dados e métodos, garantindo o controle de como eles são acessados e modificados;



# Classes imutáveis

- Uma classe imutável é aquela cujas instâncias (objetos) **não podem ser modificadas depois de criadas**;
- Todos os atributos de uma classe imutável **são declarados como private e final**;
- **Não há setters**, e qualquer alteração nos atributos requer a criação de um novo objeto;
- Classes imutáveis garantem consistência e segurança no código, evitando mudanças inesperadas no estado do objeto;
- Exemplos de classes imutáveis nativas em Java incluem **String** e classes wrappers como **Integer** e **Double**;
- Classes imutáveis são particularmente úteis em programação multithread, pois eliminam a necessidade de sincronização;





# Encapsulamento de Arrays

- Encapsulamento é a prática de **restringir o acesso direto aos atributos de uma classe**, expondo-os apenas através de métodos controlados (getters e setters);
- Quando trabalhamos com arrays ou coleções, o encapsulamento é igualmente importante;
- Arrays e coleções contêm múltiplos dados, e expô-los diretamente pode permitir modificações não controladas, causando inconsistências;
- Para encapsular arrays ou coleções, **usamos getters e setters**, garantindo que acessos e modificações sejam feitos de forma controlada;
- É importante fornecer cópias ao expor arrays ou coleções, **garantindo que o conteúdo original não seja alterado** diretamente;





# Orientação a Objetos

~~Conclusão~~





# Avançando em POO

Introdução

# Object Composition

- Composição de Objetos é um princípio da Programação Orientada a Objetos (POO), **onde um objeto é composto por outros objetos**;
- Na composição, **um objeto maior contém outros objetos menores como atributos**, combinando suas funcionalidades para formar comportamentos mais complexos;
- **A composição oferece uma alternativa à herança**, favorecendo a reutilização de código sem criar dependências rígidas entre classes;





# Herança

- **Herança** é um dos pilares da Programação Orientada a Objetos (POO);
- Permite que uma classe **herde atributos e métodos** de outra classe;
- **Cria uma relação entre as classes**, onde a subclasse é um tipo especializado da superclasse;
- **Herança promove o reuso de código**, facilitando a criação de novas classes sem duplicação de lógica;



# A classe Object

- **Object** é a superclasse de todas as classes em Java;
- Todas as classes em Java, direta ou indiretamente, **herdam da classe Object**;
- Ela define **métodos comuns que podem ser usados em qualquer classe**, como `toString()`, `equals()`, `hashCode()`, e `getClass()`;
- Esses métodos podem ser sobrescritos (**overridden**) nas classes que você criar para fornecer comportamentos específicos;





# Overriding

- Overriding (ou Sobrescrita) é um recurso da Programação Orientada a Objetos (POO) **que permite a uma subclasse fornecer uma implementação específica para um método que já está definido na superclasse;**
- O método sobrescrito na subclasse **deve ter a mesma assinatura (mesmo nome e parâmetros)** que o método da superclasse;
- **A palavra-chave @Override** é usada para indicar que um método está sendo sobrescrito;
- Overriding permite que a subclasse modifique ou especialize o comportamento herdado de uma superclasse;



# Método super

- super é uma palavra-chave em Java usada para **referenciar a superclasse de uma subclasse**;
- Através de super, uma subclasse pode:
  - Chamar o construtor da superclasse;
  - Acessar métodos e atributos da superclasse que foram sobrescritos na subclasse;
- O uso de super é comum quando a subclasse **deseja reutilizar parte do comportamento da superclasse**;





# O que é Abstraction?

- Abstração é um dos pilares fundamentais da Programação Orientada a Objetos (POO);
- Ela consiste em **ocultar os detalhes complexos de uma implementação**, expondo apenas as funcionalidades essenciais para o usuário;
- **Abstração ajuda a simplificar o uso de objetos**, escondendo a complexidade interna e focando apenas nas operações relevantes;
- Em Java, abstração é alcançada com o uso de **classes abstratas e interfaces**;



# Classes abstratas

- Uma classe abstrata em Java é uma classe que **não pode ser instanciada diretamente**;
- Ela serve como um "**modelo**" para outras classes, definindo métodos que as subclasses devem implementar
- Classes abstratas podem conter métodos:
  - **Abstratos** (sem implementação), que devem ser implementados pelas subclasses;
  - **Concretos** (com implementação), que podem ser herdados ou sobrescritos;
- A principal função de uma classe abstrata é **fornecer uma estrutura comum para subclasses**, garantindo que elas implementem certos métodos;





# Interfaces

- Uma interface em Java é um contrato que define um conjunto de métodos que uma classe deve implementar;
- Ela não fornece a implementação desses métodos, apenas suas assinaturas;
- Uma classe pode implementar múltiplas interfaces, permitindo maior flexibilidade em comparação à herança simples;
- As interfaces são ideais para definir comportamentos que podem ser compartilhados por classes não relacionadas;
- Desde Java 8, interfaces podem conter métodos concretos com a palavra-chave default;



# Múltiplas Interfaces

- Em Java, **uma classe pode implementar várias interfaces**, permitindo que ela herde comportamentos de diferentes fontes;
- **Diferente da herança**, onde uma classe pode herdar de apenas uma superclasse, com interfaces a classe pode "herdar" comportamentos de várias interfaces;
- Isso oferece maior flexibilidade ao criar classes que precisam combinar comportamentos de diferentes domínios;





# Default methods em Interfaces

- Default Methods são **métodos concretos** (com implementação) dentro de interfaces;
- Introduzidos no Java 8, eles permitem **adicionar novas funcionalidades a interfaces** existentes sem quebrar a compatibilidade com classes que já as implementam;
- Com métodos default, **você pode fornecer uma implementação padrão** que pode ou não ser **sobrescrita** pelas classes que implementam a interface;



# Classe Abstrata x Interface

- **Classe Abstrata:**

- Uma classe que não pode ser instanciada diretamente;
- Pode conter métodos abstratos (sem corpo) e métodos concretos (com corpo);
- Permite a existência de atributos e construtores;
- Serve para definir um comportamento comum, que pode ser herdado por subclasses;
- Uma classe pode herdar apenas uma única classe abstrata;

- **Interface:**

- Um contrato que define métodos que uma classe deve implementar;
- Antes do Java 8, continha apenas métodos abstratos. A partir do Java 8, pode ter métodos default e static com implementação;
- Não pode ter construtores ou atributos de instância, apenas constantes;
- Uma classe pode implementar múltiplas interfaces ao mesmo tempo;





# O que é Polimorfismo?

- **Polimorfismo** é um dos pilares da Programação Orientada a Objetos (POO);
- O termo significa "muitas formas" e **permite que uma única interface (ou tipo) seja usada para diferentes tipos de objetos**;
- Em Java, o polimorfismo permite que um objeto de uma subclasse seja tratado como um objeto de sua superclasse, e também que métodos da superclasse sejam sobrescritos pelas subclasses;
- Existem dois tipos de polimorfismo:
  - **Polimorfismo de Sobrescrita (Override)**: Quando uma subclasse fornece sua própria implementação de um método herdado da superclasse;
  - **Polimorfismo de Sobrecarga (Overload)**: Quando vários métodos têm o mesmo nome, mas com assinaturas diferentes (não será o foco aqui);



# Exercícios

- Vamos agora fazer os exercícios da seção de Avançando em POO!





# Métodos e classes final

- O que é final em Java?
  - A palavra-chave final **pode ser usada em classes, métodos e variáveis**;
  - Quando aplicada a classes e métodos, **impede modificações e sobrescritas**;
- Classes final
  - Uma classe marcada como final **não pode ser estendida**;
  - Uso: Quando você quer garantir que a implementação de uma classe não seja alterada por subclasses;
- Métodos final
  - Um método marcado como final **não pode ser sobrescrito por subclasses**;
  - Uso: Para proteger comportamentos essenciais que não devem ser alterados;
- Quando usar final?
  - **Classes final**: Use quando quiser impedir que uma classe seja herdada;
  - **Métodos final**: Use quando quiser garantir que métodos importantes não sejam sobrescritos;



# Clonagem de Objetos (Cloneable)

- Clonagem cria uma **cópia idêntica de um objeto**;
- A **interface Cloneable** habilita a clonagem em Java;
- **Dois tipos de clonagem**: superficial (shallow) e profunda (deep);
- Clonagem superficial **copia valores e referências**, mas não objetos referenciados;
- Clonagem profunda copia o objeto e todos os objetos referenciados;
- Para implementar, implemente Cloneable e sobrescreva o método clone();
- Use **super.clone()** para realizar a clonagem;
- Sem sobrescrever clone(), ocorre CloneNotSupportedException;
- Use clonagem superficial quando apenas referências são suficientes;
- Use clonagem profunda para copiar completamente os objetos referenciados;





# Reflection API

- O que é **Reflexão (Reflection API)**?
  - Reflexão é uma API que permite inspecionar e modificar o comportamento de classes, métodos e atributos em tempo de execução;
  - Com a Reflection API, você pode obter informações sobre uma classe, como seus métodos, construtores e campos, sem conhecer a classe em tempo de compilação;
- **Principais Usos da Reflexão**
  - Inspeção de classes e seus membros (métodos, construtores e campos);
  - Invocação de métodos de forma dinâmica;
  - Acessar e modificar campos privados e protegidos;
  - Criação de instâncias de classes em tempo de execução;
- **Como Usar a Reflexão?**
  - Obter a classe: Use `Class.forName()` ou `objeto.getClass()`;
  - Obter métodos, campos e construtores: Utilize os métodos da classe `Class` como `getMethods()`, `getFields()`, etc;
  - Invocar métodos: Use `Method.invoke()` para chamar métodos de forma dinâmica;





# Avançando em POO

Conclusão





# Tratamento de erros

Introdução

# O que são Exceções? (erros)

- Exceções são **eventos inesperados que interrompem a execução normal de um programa**, geralmente causados por erros de lógica ou condições imprevistas;
- Em Java, todas as exceções são objetos que herdam da **classe Throwable**;
- As exceções podem ser verificadas (checked) ou não verificadas (unchecked):
  - **Verificadas:** São verificadas pelo compilador e devem ser tratadas ou declaradas explicitamente. Ex: IOException;
  - **Não verificadas:** Ocorrências em tempo de execução, não são obrigatórias de tratar. Ex: NullPointerException;
- O tratamento de exceções permite **capturar e lidar com erros**, mantendo a estabilidade do programa e prevenindo encerramentos abruptos;





# Tempo de compilação vs Tempo de execução

- **Tempo de Compilação:** Quando o código é verificado e convertido em bytecode pelo compilador
  - Erros detectados: Sintaxe, tipos, declarações faltantes;
  - Exemplo: Esquecer um ponto e vírgula ou declarar uma variável incorretamente;
- **Tempo de Execução:** Quando o programa já compilado é executado pela máquina virtual
  - Erros detectados: Lógica, exceções não verificadas, como NullPointerException;
  - Exemplo: Divisão por zero ou acesso a um índice inexistente de array;



# Bloco try-catch

- O **bloco try-catch** permite capturar e tratar exceções que ocorrem durante a execução do programa;
- O código que pode gerar uma exceção é **colocado dentro do bloco try**;
- O **bloco catch** captura a exceção e executa o código de tratamento;
- É possível ter **múltiplos blocos catch** para tratar diferentes tipos de exceções;
- Usar try-catch evita que o programa encerre inesperadamente em caso de erro;





# Bloco finally

- O bloco finally é **usado junto com try-catch** para garantir que o código seja executado, independentemente de uma exceção ter sido lançada ou não;
- **O código no bloco finally sempre será executado**, mesmo que haja um retorno antecipado no try ou catch;
- É comumente usado para **liberar recursos**, como fechar arquivos, conexões de banco de dados ou liberar memória;
- Um bloco finally pode ser usado com ou sem um bloco catch;



# Exceções Verificadas vs. Não Verificadas

- **Exceções Verificadas:**
  - São verificadas em tempo de compilação;
  - O compilador exige que você trate ou declare essas exceções com try-catch ou throws;
  - Exemplo: IOException, SQLException;
- **Exceções Não Verificadas:**
  - Ocorrem em tempo de execução;
  - O compilador não exige que sejam tratadas;
  - Geralmente indicam erros de lógica no código;
  - Exemplo: NullPointerException, ArrayIndexOutOfBoundsException;
- **Diferença Principal:**
  - Verificadas: Tratamento obrigatório;
  - Não verificadas: Tratamento opcional, mas pode ser necessário para evitar falhas;





# Exceções são Objetos da Classe Throwable

- A palavra-chave throw é usada para **lançar uma exceção explicitamente em um bloco de código**;
- Pode ser usada para lançar **exceções verificadas e não verificadas**;
- Permite criar **exceções personalizadas** ou lançar exceções existentes em cenários específicos;
- Após lançar uma exceção com throw, o fluxo de execução é interrompido e a exceção deve ser tratada ou propagada;
- Sintaxe: `throw new ExcecaoTipo("mensagem de erro");`



# Criando Exceções Customizadas

- Exceções customizadas permitem **criar erros específicos para o contexto da sua aplicação**;
- Para criar uma exceção personalizada, você **herda de Exception (verificada) ou RuntimeException (não verificada)**;
- Você pode adicionar **mensagens de erro personalizadas** e comportamentos específicos;
- As exceções personalizadas melhoram a legibilidade do código e a tratativa de erros específicos;





# Uso do throws em Métodos

- A palavra-chave throws é usada na declaração de um método para indicar que ele pode **lançar uma ou mais exceções**;
- Métodos que lançam exceções verificadas devem declarar essas exceções usando throws;
- Exceções não verificadas **não precisam** ser declaradas com throws;
- O throws delega o tratamento da exceção para quem invoca o método, em vez de tratá-la diretamente dentro do método;
- Sintaxe: `public void metodo() throws ExcecaoTipo { ... }`



# Encadeamento de Exceções

- Encadeamento de exceções **permite associar uma exceção a outra**, para rastrear a causa original de um erro;
- Útil para identificar a causa raiz de uma exceção ao longo de **múltiplos níveis de métodos**;
- A exceção primária (externa) pode "encapsular" a exceção original (causa interna) usando o método `initCause()` ou passando a causa no construtor da exceção;
- Facilita a depuração ao preservar o histórico completo das exceções que ocorreram;
- Sintaxe: `new Excecao("Mensagem", causa)` ou `excecao.initCause(causa)`;





# Boas práticas

- **Use exceções de forma adequada:** Não abuse de exceções para controle de fluxo normal do programa;
- **Especifique o tipo de exceção:** Capture exceções específicas sempre que possível, evitando capturar a exceção genérica Exception;
- **Trate exceções localmente:** Trate a exceção no ponto mais próximo onde o erro pode ser recuperado;
- **Limpeza de recursos:** Use finally para garantir que recursos sejam liberados corretamente;
- **Encapsule exceções internas:** Utilize encadeamento de exceções para preservar a causa original;
- **Evite exceções silenciosas:** Não capture exceções sem fornecer uma mensagem de log ou saída apropriada;
- **Lance exceções específicas:** Crie exceções customizadas quando necessário para representar melhor os erros de domínio;



# Exceções multicatch

- O multicatch permite **capturar várias exceções** em um único bloco catch;
- Introduzido no Java 7, **simplifica o código** e evita **duplicação** ao capturar diferentes tipos de **exceções** com a mesma lógica de tratamento;
- Reduz o número de blocos catch, melhorando a legibilidade e manutenção do código;
- Sintaxe: `catch (Excecao1 | Excecao2 e) { ... };`





# Relançando exceção

- O re-lançamento de exceções permite capturar uma exceção, realizar alguma ação (ex.: log, limpeza de recursos) e, em seguida, **lançar novamente a exceção**;
- Útil para delegar o **tratamento completo da exceção para outro método ou camada da aplicação**;
- **Quando usar?** Quando é necessário adicionar informações ao erro, executar alguma lógica intermediária ou realizar a limpeza de recursos;
- Para preservar a pilha de execução (stack trace) original, é importante relançar a exceção sem modificá-la;
- **Sintaxe:** throw e; no bloco catch após o tratamento desejado;





# Tratamento de erros

Conclusão





# Manipulação de arquivos

Introdução

# O que é Manipulação de Arquivos em Java?

- Manipulação de arquivos em Java envolve a **leitura, escrita, e gerenciamento de arquivos e diretórios** no sistema de arquivos;
- Java fornece várias APIs para trabalhar com arquivos, como **FileReader, FileWriter, BufferedReader**, e a API **java.nio.file**;
- É possível trabalhar com arquivos de texto (como .txt) e arquivos binários (como imagens, vídeos);
- A manipulação de arquivos permite armazenar e recuperar dados, realizar operações de entrada/saída (I/O) e até serializar objetos;
- O uso adequado dessas APIs garante segurança e eficiência no gerenciamento de arquivos, **evitando problemas como corrupção de dados e vazamentos de recursos**;





# Lendo arquivos com Java

- A leitura de arquivos em Java pode ser feita usando várias classes da API, como **FileReader**, **BufferedReader**, e a API **java.nio.file**;
- **FileReader** é uma classe básica para ler dados de arquivos de texto, caracter por caracter;
- **BufferedReader** é uma classe que otimiza a leitura de arquivos, permitindo ler linhas inteiras de uma vez, o que melhora o desempenho;
- A manipulação adequada dos arquivos inclui sempre fechar o arquivo após a leitura, ou usar o **try-with-resources** para garantir o fechamento automático;
- Try-with-resources é um try catch com argumentos no try, que neste caso serão os arquivos, e eles fecham automaticamente no fim da execução;



# Escrita de arquivos em Java

- A escrita de arquivos em Java pode ser feita utilizando classes como **FileWriter** e **BufferedWriter**;
- **FileWriter** é uma classe básica que permite escrever dados de texto em arquivos, caracter por caracter;
- **BufferedWriter** é uma classe que melhora a eficiência, permitindo escrever grandes quantidades de dados de uma vez, otimizando o desempenho;
- É possível sobrescrever um arquivo existente ou adicionar novos dados ao final do arquivo (modo append);
- O uso de try-with-resources garante o fechamento automático dos arquivos após a operação, prevenindo vazamentos de recursos;





# Serialização de Objetos (Serializable)

- **Serialização** é o processo de **converter um objeto em um fluxo de bytes** para armazená-lo em um arquivo ou transmiti-lo pela rede;
- **Deserialização** é o processo inverso: recriar o objeto a partir do fluxo de bytes;
- Para serializar um objeto em Java, a classe deve implementar a **interface Serializable**;
- Vantagem: Permite salvar o estado do objeto e restaurá-lo posteriormente;
- Importante: Os campos que não devem ser serializados devem ser marcados como **transient**;
- Sintaxe: **ObjectOutputStream** para serializar e **ObjectInputStream** para deserializar;
- **Caso de uso:**
  - Um sistema de e-commerce que mantém os dados do carrinho de compras de um usuário em cache. Quando o usuário retorna ao site, o carrinho pode ser deserializado e rapidamente recuperado, permitindo que o estado do carrinho de compras seja restaurado.



# Manipulação de Arquivos Binários em Java

- Arquivos binários contêm dados que **não são diretamente legíveis como texto** (ex.: imagens, vídeos, executáveis);
- Para manipular arquivos binários, usamos classes como **FileInputStream** e **FileOutputStream**, que leem e escrevem bytes diretamente;
- **Essas classes funcionam com dados em nível de byte**, permitindo ler e escrever qualquer tipo de arquivo;
- O uso adequado dessas classes inclui sempre fechar os streams após a leitura/escrita, ou usar **try-with-resources** para garantir o fechamento automático;
- Exemplo: Copiar ou modificar arquivos binários (imagens, PDFs, etc.);





# Manipulação de Imagens

- A manipulação de imagens em Java é feita através das classes da API **java.awt** e **java.awt.image**, que permitem editar, desenhar e modificar imagens;
- O **BufferedImage** é a classe principal para armazenar imagens na memória, permitindo manipulação pixel a pixel;
- A classe **Graphics2D** fornece métodos para desenhar na imagem, como adicionar texto, formas e até outras imagens;
- Exemplos de manipulação:
  - Inserir texto ou marcas d'água;
  - Redimensionar ou rotacionar a imagem;
  - Alterar cores ou aplicar filtros;



# Manipulação de Diretórios e Arquivos

- **java.nio.file** é a API moderna para manipulação de arquivos e diretórios em Java;
- A API oferece métodos eficientes e simplificados para operações como **criação, exclusão, movimentação, e cópia de arquivos e diretórios**;
- Classes principais:
  - **Files**: Contém métodos estáticos para operações com arquivos e diretórios (criar, copiar, mover, etc.);
  - **Paths**: Representa caminhos de arquivos e diretórios de forma multiplataforma;
- Vantagens:
  - Melhor suporte a sistemas de arquivos modernos;
  - Manuseio de exceções mais consistente com a classe `IOException`;
  - Funcionalidades adicionais, como operações atômicas e manipulação de metadados de arquivos;





# Arquivos temporários

- Arquivos Temporários são arquivos criados para armazenar dados temporários e que normalmente são excluídos após o uso;
- Java oferece o método `Files.createTempFile()` para criar arquivos temporários de forma simples e segura;
- Quando usar?
  - Armazenar dados intermediários que não precisam ser mantidos após o processamento;
  - Evitar o uso excessivo de memória em operações temporárias grandes (ex: upload de arquivos);
  - Em operações que requerem um espaço de trabalho temporário durante o processamento;
- Por que usar?
  - Garantia de que o arquivo será armazenado no local apropriado para arquivos temporários do sistema;
  - Arquivos temporários podem ser automaticamente excluídos quando o programa termina;



# Manipulação de Arquivos Comprimidos (ZIP)

- Java oferece suporte nativo para arquivos ZIP através da API `java.util.zip`;
- Arquivos ZIP são usados para compactar e descompactar arquivos e diretórios, economizando espaço de armazenamento e facilitando a transferência de múltiplos arquivos;
- Classes principais:
  - **ZipOutputStream**: Usada para criar e gravar arquivos ZIP;
  - **ZipInputStream**: Usada para ler e extrair arquivos de um arquivo ZIP;
  - **ZipEntry**: Representa uma entrada (arquivo ou diretório) dentro do arquivo ZIP;
- Aplicações comuns:
  - Agrupar múltiplos arquivos para envio;
  - Reduzir o tamanho de arquivos para economizar espaço;
  - Arquivos temporários ou backups compactados;





# Leitura e escrita de arquivos CSV

- **CSV (Comma-Separated Values)** é um formato amplamente usado para armazenar dados tabulares de forma simples e legível;
- Em Java, arquivos CSV podem ser manipulados **manualmente** ou **usando bibliotecas como OpenCSV**;
- **Leitura Manual:** Simples de implementar, ideal para arquivos pequenos e estruturados de forma básica;
- **Uso de OpenCSV:** Biblioteca que facilita a leitura e escrita de arquivos CSV com suporte a operações avançadas como escape de caracteres e manipulação de cabeçalhos;





# Manipulação de arquivos

Conclusão





# Generics

Introdução

# O que são Generics?

- Generics são uma funcionalidade que **permite criar classes, interfaces e métodos que operam com tipos de dados parametrizados**;
- **Introduzidos no Java 5**, os Generics ajudam a criar código mais flexível e reutilizável, sem a necessidade de definir tipos de dados específicos;
- **Objetivo:** Permitir que uma estrutura de dados ou método trabalhe com diferentes tipos de dados, proporcionando segurança de tipos durante a compilação;
- **Generics evitam o uso excessivo de casting** e garantem que tipos incorretos não sejam atribuídos a coleções ou métodos;
- Usados amplamente em classes de **coleções como List, Set, e Map**, garantindo que os elementos armazenados sejam do tipo correto;
- **Benefícios:**
  - Segurança de tipos: Erros de tipo são detectados em tempo de compilação;
  - Reutilização de código: Um código genérico pode ser usado para diferentes tipos sem duplicação;
  - Legibilidade: O código se torna mais claro e previsível ao definir explicitamente o tipo de dados;





# Criando classes genéricas

- Classes Genéricas permitem que uma classe opere com **tipos de dados diferentes de maneira parametrizada**;
- Uma classe genérica utiliza parâmetros de tipo (**como <T>, <E>, <K, V>**) que podem ser substituídos por tipos específicos durante a criação de objetos;
- **Vantagens:**
  - Maior flexibilidade e reutilização do código sem a necessidade de duplicação para tipos diferentes;
  - Segurança de tipos em tempo de compilação, prevenindo erros relacionados a tipos;
- **Sintaxe:** A classe genérica é declarada com um parâmetro de tipo dentro de colchetes angulares (<>), como <T>, onde T pode ser qualquer nome que representa um tipo de dado;
- Classes genéricas são amplamente utilizadas nas **coleções do Java (ex: List<T>, Map<K, V>)**;
- Exemplo de Uso:
  - Uma classe genérica pode manipular qualquer tipo de dados, seja Integer, String, ou até outros objetos mais complexos, proporcionando grande versatilidade;



# Criando métodos genéricos

- Métodos Genéricos permitem que **um método opere com diferentes tipos de dados**, de forma parametrizada, semelhante a classes genéricas;
- A principal diferença é que o parâmetro de tipo é declarado no nível do método e **pode ser usado em qualquer método dentro da classe**, sem tornar a classe inteira genérica;
- **Sintaxe:** O parâmetro de tipo é declarado entre <> antes do tipo de retorno do método;
- Exemplo: <T> T metodoGenerico(T parametro);
- **Vantagens:**
  - Maior flexibilidade: O mesmo método pode manipular diferentes tipos de dados sem precisar duplicar código;
  - Segurança de tipos: Verificação em tempo de compilação garante que os tipos usados sejam consistentes;
- Métodos genéricos são muito usados em algoritmos de ordenação, busca, e manipulação de coleções;
- O uso de métodos genéricos **ajuda a evitar o uso excessivo de casting**, tornando o código mais claro e robusto;





# Tipos delimitados (bounded types)

- Tipos delimitados (Bounded Types) permitem **restringir o tipo genérico a um subtipo ou supertipo específico**, proporcionando maior controle sobre o tipo de dado aceito por uma classe ou método genérico;
- Ao usar bounded types, você pode impor que o tipo genérico **herde de uma classe específica ou implemente uma interface**;
- **Sintaxe:**
  - Para restringir um tipo genérico a uma classe ou interface, usamos o operador extends;
  - Exemplo: `<T extends Number>` – Isso significa que T deve ser uma subclasse de Number (como Integer, Double, etc.);
- **Vantagens:**
  - Permite reutilizar código genérico com maior flexibilidade e segurança, mas restringindo-o a tipos apropriados;
  - Oferece segurança de tipos em tempo de compilação;



# Coringas (Wildcards)

- Coringas (ou Wildcards) são utilizados para **tornar os tipos genéricos mais flexíveis ao permitir que parâmetros genéricos aceitem diferentes tipos sem especificá-los diretamente;**
- Um Coringa é representado pelo **símbolo ?**, e pode ser usado para definir tipos genéricos que não precisam ser exatamente um tipo específico;
- **Tipos de Coringas:**
  - Coringa Sem Limitação (?): Aceita qualquer tipo;
  - Coringa com Extensão (? extends T): Aceita T ou qualquer subtipo de T;
  - Coringa com Supertipo (? super T): Aceita T ou qualquer supertipo de T;





# O que são Collections?

- Collections são estruturas de dados **que permitem armazenar, organizar e manipular grupos de objetos em Java**;
- A Java Collections Framework oferece uma biblioteca padrão para manipulação de coleções de dados, como listas, conjuntos e mapas;
- Principais Interfaces:
  - **List**: Uma coleção ordenada de elementos que pode conter duplicatas (ex: ArrayList);
  - **Set**: Uma coleção que não permite elementos duplicados (ex: HashSet);
  - **Map**: Uma coleção de pares chave-valor, onde as chaves são únicas (ex: HashMap);
- Componentes Importantes:
  - **Iteradores**: Permitem navegar pelos elementos de uma coleção de forma segura e eficiente;
  - **Métodos utilitários**: Como sort, reverse, e shuffle, facilitam operações comuns em coleções;
- Obs: veremos alguns exemplos com Collections, e possivelmente teremos uma seção de Collections no curso =)



# Generics com Coleções (Collections)

- Generics são amplamente utilizados nas coleções do Java, como **List**, **Set**, e **Map**, para garantir segurança de tipos e evitar erros em tempo de execução;
- Usando Generics em coleções, é possível especificar o tipo de objetos que uma coleção pode armazenar, tornando o código mais previsível e seguro;
- Coleções Genéricas **evitam a necessidade de casting explícito** ao recuperar elementos e permitem que o compilador verifique se os tipos de dados inseridos são consistentes;
- Principais Coleções Genéricas:
  - **List<T>**: Armazena uma lista ordenada de elementos do tipo T;
  - **Set<T>**: Armazena uma coleção de elementos únicos do tipo T;
  - **Map<K, V>**: Armazena pares chave-valor, onde K é o tipo da chave e V é o tipo do valor;





# Uso de Generics com Interfaces

- Interfaces Genéricas permitem que o tipo de dados usado pela interface **seja definido quando a interface é implementada**;
- Isso **proporciona flexibilidade e reutilização**, permitindo que várias classes implementem a mesma interface genérica com diferentes tipos;
- **Sintaxe:**
  - Uma interface genérica é declarada com um parâmetro de tipo (`<T>`), similar a uma classe genérica;
  - Exemplo: `public interface Exemplo<T>;`
- **Exemplo de Uso:**
  - Uma interface genérica que define métodos comuns para lidar com vários tipos de dados, e classes que implementam essa interface com tipos específicos;



# Restrição Múltipla de Tipos em Generics

- Restrição Múltipla de Tipos permite **que um parâmetro genérico seja limitado a múltiplas classes e interfaces**, garantindo que ele atenda a várias condições simultaneamente;
- Sintaxe:
  - **T extends ClassA & InterfaceB**: Define que o tipo T deve ser uma subclasse de ClassA e também deve implementar a InterfaceB;
  - **Apenas uma classe pode ser especificada** (pois o Java não suporta herança múltipla), mas você pode listar várias interfaces;
- **Exemplo de Uso**:
  - Um método genérico que exige que o tipo seja uma subclasse de uma classe específica e, ao mesmo tempo, implemente uma ou mais interfaces;







# Generics

Conclusão



# Collections

Introdução



# Introdução às Collections

- O que são Collections?
  - Collections são estruturas de dados que permitem **armazenar, organizar e manipular grupos de objetos** em Java;
  - Elas **facilitam a manipulação de conjuntos de dados**, como listas, conjuntos, filas e mapas;
- Importância das Collections na Programação
  - Proporcionam uma forma eficiente e flexível de manipular grandes quantidades de dados;
  - As Collections eliminam a necessidade de criar estruturas de dados manualmente;
  - Oferecem uma série de métodos úteis para **ordenar, buscar, filtrar, adicionar e remover** dados;
- Java Collections Framework (JCF)
  - A Java Collections Framework é uma biblioteca padrão do Java que oferece um conjunto de classes e interfaces para trabalhar com coleções de dados;
  - Fornece implementações de estruturas de dados comuns como **List, Set, Map, e Queue**;
  - As coleções são altamente flexíveis e genéricas, permitindo que tipos de dados sejam parametrizados com Generics;



# Interfaces Principais das Collections

- List
  - Armazena elementos **ordenados de acordo com a ordem de inserção**, aceita elementos duplicados;
  - Acesso rápido a elementos por índice;
  - **Implementações:** ArrayList, LinkedList;
  - Aplicação: Ideal quando você precisa manter a ordem dos elementos e acessar itens por índice;
- Set
  - **Armazena uma coleção de elementos únicos (não permite duplicatas)**, não garante a ordem de inserção, exceto com implementações específicas;
  - **Implementações:** HashSet (não ordenado), LinkedHashSet (mantém a ordem de inserção), TreeSet (ordenado automaticamente);
  - Aplicação: Garantir que não existam duplicatas e a ordem dos elementos não é importante;
- Map
  - **Armazena pares chave-valor**, onde cada chave é única, não permite chaves duplicadas;
  - **Implementações:** HashMap, LinkedHashMap, TreeMap;
  - Aplicação: Ideal para armazenar dados com uma associação direta entre chave e valor, como dicionários ou catálogos;





# Trabalhando com List

- Introdução à Interface List
  - A interface List é usada para **armazenar uma sequência ordenada de elementos**;
  - **Permite elementos duplicados** e o acesso aos itens por índice;
  - Oferece métodos eficientes para manipulação de elementos como adicionar, remover e modificar;
- Tipos de List
- **ArrayList:**
  - Implementa List usando um array dinâmico;
  - Acesso rápido por índice, porém operações de inserção/remoção são mais lentas, especialmente em grandes listas;
- **LinkedList:**
  - Implementa List como uma lista duplamente encadeada;
  - Inserção/remoção rápida, especialmente em qualquer posição, mas o acesso por índice é mais lento;



# ArrayList vs LinkedList

- **ArrayList**

- Acesso Rápido: Acesso eficiente por índice.
- Inserção/Remoção Lenta: Operações de inserção e remoção são lentas em grandes listas, especialmente no início/meio da lista.
- Melhor para leitura frequente: Ideal quando há mais operações de leitura do que de modificação.

- **LinkedList**

- Inserção/Remoção Rápida: Operações de inserção e remoção são rápidas, especialmente em qualquer posição da lista.
- Acesso Lento: Acesso por índice é mais lento devido à estrutura encadeada.
- Melhor para inserção/remoção frequente: Ideal para cenários com muitas modificações nos dados.





# Trabalhando com Set

- Introdução à Interface Set
  - Set é uma coleção **que não permite duplicatas**;
  - **Os elementos em um Set não possuem uma ordem definida**, exceto em implementações específicas;
  - Ideal para armazenar dados únicos onde a ordem não é tão importante;
- Tipos de Set
  - **HashSet**: Não garante a ordem dos elementos, mas é eficiente para operações como adicionar e verificar elementos;
  - **LinkedHashSet**: Mantém a ordem de inserção dos elementos;
  - **TreeSet**: Mantém os elementos ordenados de acordo com a ordem natural ou um Comparador personalizado;
- Quando Usar Set vs List
  - Use Set quando não quiser duplicatas e a ordem dos elementos não for prioritária;
  - Use List quando a ordem dos elementos for importante e duplicatas forem permitidas;



# Trabalhando com Map

- Introdução à Interface Map
  - Map é uma coleção que armazena pares chave-valor, onde cada chave é única;
  - Uma chave pode ter apenas um valor associado, mas os valores podem se repetir;
  - Ideal para associações entre chave e valor, como dicionários ou catálogos;
- Tipos de Map
  - **HashMap:** Não mantém a ordem de inserção. Muito eficiente para busca e inserção;
  - **LinkedHashMap:** Mantém a ordem de inserção das chaves;
  - **TreeMap:** Mantém as chaves ordenadas de acordo com a ordem natural ou um Comparator;





# Iterando sobre Collections

- Maneiras de Iterar sobre Coleções
  - **For-each Loop:** Simples e legível, ideal para leitura de elementos;
  - **Iterator:** Permite navegação sobre coleções, com a capacidade de remover elementos de forma segura durante a iteração;
  - **ListIterator:** Estende o Iterator, permitindo navegar em ambas as direções (para frente e para trás) em coleções baseadas em lista (como ArrayList);
- Por que usar o Iterator?
  - Permite a remoção segura de elementos enquanto você percorre a coleção, o que não é possível diretamente com um for-each;
  - Evita exceções como ConcurrentModificationException ao modificar a coleção enquanto a percorre;



# Collections Imutáveis

- O que são coleções imutáveis?
  - Coleções imutáveis são aquelas cujo **conteúdo não pode ser modificado** (adicionar, remover ou atualizar elementos);
  - Após a criação, não é possível alterar os dados na coleção;
  - **Evita modificações acidentais** e melhora a segurança de dados em ambientes onde múltiplas threads estão acessando os dados;
- Quando usar coleções imutáveis?
  - Quando você quer garantir que os dados de uma coleção não serão alterados após sua criação;
  - Melhor prática em cenários de concorrência, onde múltiplas threads podem acessar os dados simultaneamente;
  - Em APIs que retornam dados para consumidores e não devem ser modificados;
- Criando Coleções Imutáveis
  - **Collections.unmodifiableList()**: Cria uma versão imutável de uma lista existente;
  - **List.of()**, **Set.of()**, **Map.of()**: Métodos da API Java 9+ que criam coleções imutáveis diretamente;





# Stream em Collections

- O que é Stream?
  - Stream é uma API introduzida no Java 8 para **processamento de dados em coleções** de forma declarativa;
  - Permite realizar operações funcionais como **filtrar, mapear e reduzir coleções**, sem modificar a coleção original;
- Operações Principais em Stream
  - **Intermediárias:** Executadas em uma coleção, retornam um novo Stream (ex.: filter(), map(), sorted());
  - **Finais:** Executadas ao final do pipeline de operações, retornam um resultado ou efeito colateral (ex.: collect(), forEach(), reduce());
- A seguir veremos vários destes métodos de Stream em ação!



# Filtragem de Collections

- O que é filtragem de Collections?
  - Filtragem é o processo de **selecionar elementos de uma coleção com base em uma condição** específica;
  - Permite extrair subconjuntos de dados **sem modificar a coleção original**;
- Quando usar filtragem?
  - Quando você deseja trabalhar com um subconjunto dos dados em vez de toda a coleção;
  - Para limitar ou refinar dados em coleções grandes antes de realizar operações adicionais;
- Métodos para Filtragem de Collections
  - **Streams (Java 8+)**: Usando filter() em Streams para aplicar condições de filtragem;
  - **Iteração manual**: Usar loops para filtrar dados de uma coleção sem Streams;





# Busca de Collections

- O que é busca em Collections?
  - Busca é o processo de **encontrar um elemento dentro de uma coleção**, com base em uma condição ou valor específico;
  - **É utilizada para verificar se um determinado dado existe na coleção** ou para localizar itens que atendem a critérios específicos;
- Quando usar busca?
  - Para verificar se um valor específico existe na coleção;
  - Para encontrar um ou mais elementos que atendam a determinadas condições;
- Métodos de busca em Collections
  - **contains()**: Verifica se a coleção contém um elemento específico;
  - **Streams (Java 8+)**: Usando `findFirst()`, `findAny()`, ou `filter()` para buscar elementos com base em condições;
  - **Iteração manual**: Usar loops para buscar manualmente elementos em coleções;



# Utilizando o Map

- O que é map()?
  - map() é uma operação intermediária em Streams **que aplica uma função a cada elemento de uma coleção**, transformando-os em um novo valor ou tipo;
  - **O resultado é um novo Stream com os elementos transformados**, sem modificar a coleção original;
- Quando usar map()?
  - Use map() quando você precisar transformar os elementos de uma coleção para outra forma ou estrutura;
  - Exemplo: Converter uma lista de números em seus quadrados, ou transformar objetos complexos em um de seus atributos;
- Sintaxe
  - **stream.map(Function)**: Aplica a função especificada a cada elemento do Stream;





# Utilizando o Reduce

- O que é `reduce()`?
  - `reduce()` é uma operação final em Streams que reduz os elementos de um Stream a um único valor;
  - É usado para **agregar os elementos de uma coleção de maneira cumulativa**, como somar, multiplicar, ou concatenar;
- Quando usar `reduce()`?
  - Quando você precisa combinar ou agregar os elementos de uma coleção em um único resultado;
  - Exemplo: Somar todos os números em uma lista, calcular o produto de números, ou concatenar strings;
- Sintaxe
  - **`stream.reduce(identity, BinaryOperator)`**: Reduz os elementos aplicando a operação fornecida;
  - **`identity`**: Valor inicial da operação de redução;
  - **`BinaryOperator`**: Função que especifica como dois elementos devem ser combinados;



# Ordenação Personalizada com Comparator

- O que é Comparator?
  - A interface Comparator é usada para **definir uma ordenação personalizada** de objetos;
  - Permite criar comparações complexas entre objetos **sem modificar a classe original**;
- Vantagens do Comparator
  - Flexibilidade para criar **múltiplas ordenações** para o mesmo tipo de objeto;
  - Facilita a ordenação por múltiplos atributos, como ordenar por nome e, em seguida, por idade;
- Métodos principais
  - **compare(T o1, T o2)**: Compara dois objetos e retorna um valor negativo, zero, ou positivo dependendo da ordem;
  - **thenComparing()**: Permite criar uma ordenação aninhada, aplicando múltiplos critérios de comparação;





# Uso avançado de Streams

- O que é flatMap()?
  - flatMap() é usado para transformar **coleções de coleções** em uma **única coleção**, "achatando" o resultado;
  - Permite processar estruturas aninhadas, como listas dentro de listas;
- Pipeline de Streams Avançado
  - **Combinar múltiplas operações intermediárias** como:
    - filter(): Filtra os elementos com base em uma condição;
    - map(): Transforma cada elemento;
    - sorted(): Ordena os elementos;
- Operações são encadeadas em um pipeline de Stream, **que só é executado quando uma operação terminal é chamada**;



# Manipulação de Coleções com Collectors

- O que é Collectors?
  - Collectors é uma classe utilitária que oferece várias **operações de redução em coleções**;
  - Permite agregar, agrupar, contar, e transformar coleções de forma eficiente;
- Operações Principais com Collectors
  - **collect()**: Operação final que converte o resultado de um Stream em uma coleção ou outro tipo de dado;
  - **Agrupamento (groupingBy)**: Agrupa elementos com base em uma chave;
  - **Particionamento (partitioningBy)**: Divide a coleção em duas partes com base em uma condição;
  - **Agregação (counting(), summingInt())**: Conta ou soma os elementos da coleção;







# Collections

Conclusão



# Expressões Regulares

Introdução



# O que são Expressões Regulares?

- **Expressões Regulares (Regex)** são padrões utilizados para procurar e manipular texto;
- Elas permitem **verificar se um padrão de caracteres aparece em uma string**, como números, letras ou símbolos;
- São amplamente usadas para validação, extração e substituição de texto em diversas linguagens de programação;
- A regex **usa metacaracteres e sequências especiais para definir padrões complexos**, como `.` (qualquer caractere), `*` (zero ou mais ocorrências) e `[]` (conjunto de caracteres);
- Em Java, expressões regulares são manipuladas com as classes `Pattern` e `Matcher` do pacote `java.util.regex`;
- Regex é muito utilizada para validações como verificar e-mails, números de telefone, senhas, e formatos de datas;
- Elas podem variar de simples a extremamente complexas, dependendo do padrão a ser definido;



# Sintaxe básica das Regex

- Metacaracteres são símbolos especiais que representam padrões complexos nas expressões regulares:
  - `.`: Representa qualquer caractere, exceto nova linha;
  - `*`: Indica zero ou mais ocorrências do caractere anterior;
  - `+`: Indica uma ou mais ocorrências do caractere anterior;
  - `?`: Indica zero ou uma ocorrência do caractere anterior (opcional);
  - `[]`: Define um conjunto de caracteres. Ex: `[abc]` corresponde a "a", "b", ou "c";
  - `^`: Representa o início da string;
  - `$`: Representa o fim da string;
- Diferença entre caracteres literais e metacaracteres:
  - **Caractere literal**: Representa o próprio caractere, como "a", "1", ou "@";
  - **Metacaractere**: Tem um significado especial na regex, como `.` ou `*`. Para usá-los como literais, é preciso escapá-los com uma barra invertida (`\`). Ex: `\.` corresponde a um ponto literal;





# Quantificadores

- Quantificadores definem quantas vezes um padrão pode ocorrer:
- **\*** (**zero ou mais**): O caractere anterior pode aparecer zero ou mais vezes. Exemplo: `a*` corresponde a "", "a", "aa", "aaa", etc.
- **+** (**uma ou mais**): O caractere anterior deve aparecer pelo menos uma vez. Exemplo: `a+` corresponde a "a", "aa", "aaa", mas não "".
- **?** (**zero ou uma**): O caractere anterior pode aparecer zero ou uma vez. Exemplo: `a?` corresponde a "", "a".
- **{n}**: O caractere anterior deve aparecer exatamente n vezes. Exemplo: `a{3}` corresponde a "aaa".
- **{n,}**: O caractere anterior deve aparecer pelo menos n vezes. Exemplo: `a{2,}` corresponde a "aa", "aaa", etc.
- **{n,m}**: O caractere anterior deve aparecer entre n e m vezes. Exemplo: `a{2,4}` corresponde a "aa", "aaa", ou "aaaa".



# Âncoras e Fronteiras

- Âncoras são utilizadas para definir posições dentro de uma string:
  - `^`: Indica o início de uma string;  
Exemplo: `^a` encontra strings que começam com "a";
  - `$`: Indica o fim de uma string;  
Exemplo: `a$` encontra strings que terminam com "a";
- Fronteiras de Palavras e Não-Fronteiras:
  - `\b`: Representa uma fronteira de palavra (entre um caractere de palavra e um não caractere de palavra);  
Exemplo: `\bword\b` encontra "word" isolada, mas não "password";
  - `\B`: Representa uma não fronteira de palavra (quando não há uma fronteira de palavra);  
Exemplo: `\Bword` encontra "password", mas não "word" isolada;





# Grupos e Captura

- **Grupos e Captura:**
  - Usam parênteses () para agrupar e capturar partes específicas de uma expressão regular;
  - Cada grupo capturado pode ser reutilizado ou referenciado;
- **Backreferences:**
  - Permite reutilizar um grupo capturado dentro da própria expressão com \1, \2, etc;
  - Útil para encontrar padrões repetidos ou substituir partes de uma string;
- **Principais Usos de Grupos:**
  - **Agrupamento:** Controla a ordem de aplicação dos operadores, como \* ou +;
  - **Captura:** Extrai e reutiliza subsequências correspondentes na expressão;



# Avançando em Pattern e Matcher

- Principais Métodos da Classe Pattern:
  - **compile(String regex):** Compila uma expressão regular em um padrão;
  - **matches(String regex, CharSequence input):** Verifica se a string corresponde totalmente ao padrão;
  - **split(String input):** Divide uma string com base em um padrão;
- Principais Métodos da Classe Matcher:
  - **find():** Busca subsequências que correspondem ao padrão;
  - **group():** Retorna o valor da última correspondência encontrada;
  - **replaceAll(String replacement):** Substitui todas as correspondências em uma string;
  - **matches():** Verifica se a string corresponde totalmente ao padrão;
  - **start() e end():** Retorna o índice de início e fim da correspondência encontrada;





# Validação de Strings

- Validação de Strings com Regex:
  - Expressões regulares (regex) são usadas **para validar se uma string segue um formato específico**;
  - Dados comuns validados com regex incluem e-mails, números de telefone, senhas, entre outros;
- Validação de E-mail:
  - Um e-mail válido segue o formato: user@dominio.com;
  - Regex para e-mails: `^[\w.-]+@[ \w.-]+\.[a-z]{2,}$`;
- Validação de Números de Telefone:
  - Um número de telefone pode ter diferentes formatos: (XX) XXXX-XXXX ou XXXXX-XXXX;
  - Regex para telefone: `^\((?\d{2})\)\)?[- ]?(?\d{4,5})[- ]?(?\d{4})$`;



# Expressões Regulares Avançadas

- **Lookaheads:**
  - Verifica se **há um padrão à frente** sem consumi-lo na correspondência;
- **Sintaxe:**
  - Lookahead positivo: `(?=...)`
  - Lookahead negativo: `(?!...)`
  - `\d+(?=\$)`: Encontra números seguidos de um símbolo de dólar (\$), mas não inclui o dólar na correspondência.
- **Lookbehinds:**
  - Verifica se **há um padrão antes** sem consumi-lo na correspondência;
- **Sintaxe:**
  - Lookbehind positivo: `(?<=...)`
  - Lookbehind negativo: `(?<!...)`
- **Exemplo:**
  - `(?<=#)\w+`: Encontra palavras que seguem imediatamente um símbolo de #, mas não inclui o # na correspondência;





# Regex para Manipulação de Datas e Horários

- **Validação de Datas com Regex:**
  - Formato DD/MM/YYYY:
  - `^([0-2][0-9]|3[01])/(0[1-9]|1[0-2])/([0-9]{4})$`
  - Valida dias de 01 a 31, meses de 01 a 12, e anos de 4 dígitos;
- **Validação de Horários com Regex:**
  - Formato HH:MM:SS
  - `^([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]$`
  - Valida horas de 00 a 23, minutos de 00 a 59, e segundos de 00 a 59;



# Regex com Flags (Opções Especiais)

- O que são Flags:
  - Flags são opções especiais que **alteram o comportamento padrão das expressões regulares em Java**;
  - São usadas para modificar a forma como o regex é processado, permitindo correspondências mais flexíveis;
- Principais Flags em Java:
  - **Pattern.CASE\_INSENSITIVE**: Ignora diferenças entre maiúsculas e minúsculas;
  - **Pattern.MULTILINE**: Trata o início (^) e fim (\$) como correspondências por linha em vez de toda a string;
  - **Pattern.DOTALL**: Permite que o caractere . corresponda a todas as linhas, inclusive quebras de linha;
  - **Pattern.COMMENTS**: Permite adicionar comentários e ignorar espaços em branco no regex, tornando-o mais legível;
- **Sintaxe para Usar Flags**:
  - `Pattern.compile(String regex, int flags)`: Compila uma expressão regular com a flag fornecida;







# Expressões Regulares

Conclusão



# Annotations

Introdução



# O que são Annotations?

- Annotations são metadados que fornecem informações adicionais ao compilador e à máquina virtual;
- Elas não afetam diretamente a execução do código, mas podem ser usadas para modificar comportamentos durante a compilação, tempo de execução ou até mesmo em frameworks;
- **Funções das Annotations:**
  - Sinalizam instruções especiais para o compilador;
  - Auxiliam no tratamento de erros e na documentação do código;
  - Automatizam tarefas repetitivas por meio de ferramentas de processamento;
- **Tipos de Annotations:**
  - Predefinidas: Como `@Override`, `@Deprecated`, e `@SuppressWarnings`;
  - Customizadas: Desenvolvedores podem criar suas próprias anotações para atender a necessidades específicas;



# Anotações Predefinidas em Java

- **@Override:**
  - Indica que um método está sobrescrevendo um método da superclasse;
  - Fornece verificação em tempo de compilação para garantir que a sobrescrita seja válida;
- **@Deprecated:**
  - Marca métodos, classes ou variáveis que estão obsoletos e podem ser removidos em versões futuras;
  - Gera um aviso no compilador ao usar elementos marcados com essa anotação;
- **@SuppressWarnings:**
  - Instrui o compilador a suprimir certos tipos de avisos que ele geraria;
  - Pode ser configurado para ignorar diferentes tipos de avisos, como unchecked, deprecation, etc;
- **Por que usar?**
  - @Override: Garante que a sobrescrita seja correta;
  - @Deprecated: Sinaliza elementos obsoletos, orientando o desenvolvedor a evitar seu uso;
  - @SuppressWarnings: Permite limpar o código de avisos desnecessários, mantendo o foco em alertas importantes;





# Criando Annotations

- O que são Anotações Customizadas?
  - São **anotações definidas pelo desenvolvedor** para fornecer informações ou comportamentos **específicos** no código;
  - Podem ser usadas em **classes, métodos, parâmetros, variáveis e campos**;
- Elementos de uma Anotação Customizada:
  - **@Target:** Define onde a anotação pode ser aplicada (classe, método, etc.);
  - **@Retention:** Define quando a anotação está disponível (em tempo de compilação, execução ou apenas no código fonte);
  - **@Documented:** Indica que a anotação deve ser incluída na documentação;
  - **@Inherited:** Permite que a anotação seja herdada por subclasses;
- Sintaxe de uma Anotação Customizada:
  - Anotações são definidas com **@interface**;
  - Elas **podem conter parâmetros** com valores padrão ou obrigatórios;



# Validação de Campos com Annotation

- Anotação Customizada para Validação de Campos:
  - **Objetivo:** Criar uma anotação customizada para validar se um campo de uma classe atende a uma determinada regra (ex.: não pode ser nulo ou vazio);
- Elementos Importantes:
  - **@Target(ElementType.FIELD):** Define que a anotação pode ser aplicada a campos de uma classe;
  - **@Retention(RetentionPolicy.RUNTIME):** A anotação está disponível em tempo de execução; permitindo sua verificação por meio de reflection;
  - **@interface:** Define a anotação customizada com parâmetros específicos;





# Processadores de Annotations

- O que são Processadores de Anotações?
  - São ferramentas que **permitem analisar e processar anotações customizadas em tempo de compilação ou execução**;
  - Os processadores são usados para **automatizar tarefas com base nas informações fornecidas pelas anotações**, como gerar código, realizar validações e otimizar o comportamento do programa;
- Tipos de Processamento:
  - **Em Tempo de Execução (Runtime)**: Usando Reflection para verificar e processar anotações;
  - **Em Tempo de Compilação (Compile-time)**: Usando Annotation Processors (processadores de anotação) para analisar e manipular anotações durante a compilação;





# Annotations

Conclusão





# JavaFx

## Introdução

# O que é JavaFX?

- **JavaFX** é uma biblioteca de software para criar interfaces gráficas de usuário (GUIs) em Java;
- **Substitui o antigo Swing** como principal biblioteca gráfica no Java;
- Permite o desenvolvimento de aplicações desktop e **RIAs (Rich Internet Applications)** com Java;
- Oferece suporte a diversos tipos de componentes gráficos: **botões, tabelas, gráficos**, etc;
- Utiliza uma abordagem baseada em cena e nó (**Scene Graph**), onde os elementos gráficos são organizados em uma estrutura de árvore;
- **Possui integração com CSS** para estilização e com FXML, um formato XML para descrever a interface;
- Suporta animações, multimídia e a **criação de gráficos 2D e 3D**;
- **Funciona em diferentes plataformas**: Windows, Mac e Linux;
- Ideal para o desenvolvimento de aplicações, como ferramentas de visualização, jogos simples e softwares empresariais;





# Como instalar e executar o JavaFX

- Documentação oficial de instalação: [clique aqui](#).
- Primeiramente precisamos **baixar o JavaFX no site oficial**, e extrair os arquivos para uma pasta;
- Depois vamos abrir o VS Code e **criar uma pasta para a seção**;
- Acesse a opção Help -> Show All Commands -> Create Java Project, e **selecione No build tools**;
- **Selecione a pasta pai (pasta da seção)**, e nomeie o projeto, por exemplo: HelloWorldFX;
- Agora em **Java Projects**, podemos adicionar os arquivos de JavaFX em Referenced Libraries;
- **Delete o arquivo criado automaticamente** com o nome de App.java;
- **Crie uma pasta** com o nome do projeto em src, exemplo: helloworldfx;
- Adicione os arquivos **Main, fxml e Controller**, que são fornecidos pela documentação;
- Vá até a aba **Run and Debug do VS Code**, e selecione a opção Create json file;
- Configure o valor de **vmArgs** no launch.json, conforme a documentação indica;
- Adapte o arquivo **Main e .fxml** para o nome que foi criado, pois está com o nome da documentação;
- Altere a mensagem do **setText** e execute o arquivo! =)



# A Classe Application e o Método Start

- **Estrutura Básica de um Aplicativo JavaFX**
  - Todo aplicativo JavaFX herda da classe Application;
  - O ciclo de vida do aplicativo começa ao executar o método launch();
- **Como Usar a Classe Application**
  - A classe Application fornece a base para criar a interface gráfica;
  - O método abstrato start() precisa ser implementado para inicializar a interface;
- **O Ciclo de Vida do Método start()**
  - O método start(Stage primaryStage) é chamado após o launch();
  - É responsável por definir o conteúdo principal da janela (Stage);
- **O ciclo de vida do aplicativo JavaFX:**
  - **Iniciação:** O método launch() é chamado;
  - **Inicialização:** O método start() é chamado para configurar a interface;
  - **Execução:** A interface é exibida e interage com o usuário;
  - **Encerramento:** Quando o usuário fecha a janela ou o sistema encerra o aplicativo;





# Stage e Scene em JavaFX

- **Stage:** Representa a janela principal da aplicação JavaFX
  - O Stage é a janela que pode ser exibida na tela, contendo o título, o tamanho e o conteúdo;
  - Pode ser fechado, maximizado, minimizado, e várias janelas (stages) podem ser criadas;
- **Scene:** Representa o conteúdo dentro de um Stage
  - A Scene contém os elementos visuais (botões, layouts, textos) que serão exibidos ao usuário;
  - Define o tamanho da área de exibição e pode conter um layout com vários componentes;
- **Relacionamento entre Stage e Scene**
  - O Stage é a "janela", e a Scene é o "conteúdo" dentro dessa janela;
  - Uma Stage pode conter apenas uma Scene de cada vez, mas essa Scene pode ser trocada dinamicamente durante a execução da aplicação;
- Vamos ver na prática!
- Obs: **todo novo arquivo deve ter configurado o vmArgs em launch.json**,  
e o nome do arquivo aparece em launch quando executamos ele via Run Java;



# Layouts em JavaFX

- **O que são Layouts em JavaFX?**
  - Layouts são gerenciadores responsáveis por organizar a posição e o tamanho dos componentes (botões, labels, etc.) dentro de uma interface;
  - Controlam como os componentes se ajustam ao redimensionamento da janela;
  - Garantem consistência na apresentação da interface, independentemente do tamanho da tela ou dispositivo;
- **Por que usar Layouts?**
  - Evitam a necessidade de posicionar cada componente manualmente;
  - Adaptam os componentes de forma eficiente para diferentes resoluções;
  - Facilitam o design de interfaces complexas de maneira estruturada e escalável;
- **Tipos de layout:**
  - VBox, HBox, BorderPane, GridPane;





# VBox e HBox em JavaFX

- **VBox:** Organiza os componentes em uma coluna, de cima para baixo
  - Útil para empilhar elementos como botões, labels e caixas de texto em uma disposição vertical;
- **HBox:** Organiza os componentes em uma linha, da esquerda para a direita
  - Ideal para criar layouts onde os componentes estão lado a lado, como uma barra de ferramentas ou opções;
- **Espaçamento:** Controla o espaço entre os componentes
  - Em VBox, o espaçamento é vertical (entre os elementos empilhados);
  - Em HBox, o espaçamento é horizontal (entre os elementos lado a lado);
- **Alinhamento:** Define como os componentes serão alinhados dentro do contêiner
  - Pode ser definido para alinhar à esquerda, direita, centro ou ajustado ao conteúdo;



# BorderPane em JavaFX

- O BorderPane organiza os componentes em cinco regiões:
  - **Top:** Parte superior, geralmente usada para cabeçalhos ou menus;
  - **Bottom:** Parte inferior, comum para rodapés ou botões de ação;
  - **Left:** Lado esquerdo, pode conter menus ou painéis de navegação;
  - **Right:** Lado direito, usado para informações adicionais ou detalhes;
  - **Center:** Área central, onde fica o conteúdo principal da aplicação;
- Cada região pode conter **um único componente ou layout**. Esse componente pode ser um botão, texto, ou até mesmo outro layout;
- Os métodos `setTop()`, `setBottom()`, `setLeft()`, `setRight()` e `setCenter()` são usados para definir os componentes em cada região do BorderPane;





# GridPane em JavaFX

- **GridPane** é um layout que organiza os componentes em uma grade de linhas e colunas;
- Ideal para criar formulários, tabelas e layouts com alinhamento preciso;
- Cada célula da grade **pode conter um único componente** (botões, campos de texto, etc.);
- Cada componente é colocado em uma célula usando o método `add(node, columnIndex, rowIndex)`, onde:
  - **columnIndex**: Posição da coluna onde o componente será inserido;
  - **rowIndex**: Posição da linha onde o componente será inserido;
- Também é possível fazer o componente ocupar várias colunas ou linhas com `setColumnSpan()` e `setRowSpan()`;



# StackPane e AnchorPane em JavaFX

- **StackPane** empilha os componentes uns sobre os outros;
- Útil quando você deseja **sobrepor elementos**, como colocar texto sobre uma imagem;
- O último componente adicionado é exibido no topo;
- **Posicionando Elementos com AnchorPane**
  - AnchorPane permite ancorar (fixar) componentes nas bordas da janela;
  - Cada componente pode ser ancorado em uma ou mais bordas (top, bottom, left, right);
  - É útil para layouts onde você deseja que os elementos permaneçam fixos ou redimensionados conforme a janela muda de tamanho;





# Botões e Labels em JavaFX

- **O que são Controles?**
  - Controles são componentes interativos usados em interfaces gráficas, como botões, caixas de texto, checkboxes, entre outros;
  - Permitem que o usuário interaja com a aplicação de maneira intuitiva;
- **Exemplos comuns de controles incluem:**
  - Button (Botão)
  - Label (Rótulo de texto)
  - TextField (Campo de texto)
- **Criando e Configurando Botões e Labels**
  - **Button:** Representa um botão clicável
    - Pode ser configurado com texto, ícones, ou ambos;
    - Ação de clique é configurada com um EventHandler;
  - **Label:** Exibe um rótulo de texto não interativo
    - Usado para mostrar informações estáticas ou descrever outros controles;



# TextField e TextArea em JavaFX

- **Criando Campos de Texto e Áreas de Texto**
  - **TextField:** Campo de texto de linha única usado para entradas simples, como nomes ou números. Ideal para entradas curtas e simples;
  - **TextArea:** Campo de texto de múltiplas linhas, usado para entradas longas, como descrições ou comentários. Permite ao usuário inserir texto em várias linhas;
- **Limite de Caracteres:**
  - Pode-se limitar o número de caracteres que o usuário pode digitar em um TextField ou TextArea utilizando eventos de texto;
- **Interatividade:**
  - Ambos os componentes podem reagir a eventos de teclado, como pressionar teclas, inserção de texto, e podem ser usados para captura de dados;





# CheckBox e RadioButton em JavaFX

- **CheckBox:**
  - Representa uma caixa de seleção independente;
  - O usuário pode selecionar ou desmarcar várias opções simultaneamente;
  - Ideal para quando várias opções podem ser escolhidas;
- **RadioButton:**
  - Usado quando apenas uma opção pode ser selecionada dentro de um grupo de opções;
  - Para funcionar corretamente, os RadioButtons devem ser agrupados usando um ToggleGroup;
  - Útil quando o usuário deve escolher uma única opção de várias;
- **ToggleGroup:**
  - Um ToggleGroup é usado para agrupar RadioButtons, garantindo que apenas uma opção dentro do grupo possa ser selecionada por vez;
  - Todos os RadioButtons dentro de um ToggleGroup se comportam como uma escolha única (ou seja, só pode haver um selecionado ao mesmo tempo);



# ComboBox e ListView em JavaFX

- **ComboBox:** Um controle de seleção em formato de menu suspenso que permite ao usuário escolher uma opção de uma lista
  - Permite que o usuário selecione uma única opção;
  - É possível adicionar opções (itens) dinamicamente ou de forma estática;
- **ListView:** Um controle que exibe uma lista de itens em uma coluna
  - O usuário pode selecionar um ou mais itens da lista;
  - Suporta seleção simples e múltipla;
  - Permite adicionar, remover e manipular os itens da lista durante a execução do programa;





# Eventos em JavaFx

- **O que são eventos em JavaFX?**
  - Eventos são ações realizadas pelo usuário ou pelo sistema que a aplicação pode "ouvir" e responder;
- **Exemplos de eventos:**
  - **Clique de botão:** Quando um botão é pressionado pelo usuário;
  - **Movimento do mouse:** O movimento ou clique do mouse em uma área da interface;
  - **Teclado:** Pressionamento de uma tecla ou combinação de teclas;
  - **Mudança de foco:** Quando um campo de texto ganha ou perde foco;
  - Cada evento gera um objeto de evento que contém informações sobre a ação realizada;
- **Modelo de eventos em JavaFX segue o padrão Delegação de Eventos, onde o evento é capturado e encaminhado para o manipulador correto:**
  - **Origem do evento:** O componente que gera o evento, como um botão;
  - **Objeto de evento:** A informação sobre o evento (ex.: MouseEvent, ActionEvent);
  - **Listener/Handler (Manipulador de Evento):** Um bloco de código que responde ao evento;
- **Event Handling (Manipulação de Eventos):**
  - O listener ou handler é associado ao componente (ex.: botão) que deseja capturar o evento;
  - Quando o evento ocorre, o listener é chamado para executar o código associado;



# Eventos de Clique em JavaFX

- Eventos de clique são capturados quando o usuário interage com um botão, pressionando-o e soltando-o;
- O evento mais comum é o **ActionEvent**, que é gerado ao clicar em um botão;
- Para lidar com eventos de clique, usamos **event handlers** ou **listeners**, que são blocos de código associados ao componente (botão);
- **Passos para Capturar um Evento de Clique:**
  - Criar o componente interativo (ex.: Button);
  - Adicionar um EventHandler usando o método `setOnAction()` do botão;
  - Implementar o código que deve ser executado quando o evento de clique ocorre;





# Eventos de Teclado e Mouse em JavaFX

- **Eventos de Teclado:**
  - Capturam ações do usuário relacionadas ao teclado, como pressionar, soltar ou digitar uma tecla;
- **Principais eventos:**
  - **KeyPressed:** Quando uma tecla é pressionada;
  - **KeyReleased:** Quando uma tecla é liberada;
  - **KeyTyped:** Quando uma tecla é digitada (geralmente usado para texto);
- **Eventos de Mouse:**
  - Capturam interações do usuário com o mouse, como movimento e cliques;
- **Principais eventos:**
  - **MouseClicked:** Quando o mouse é clicado;
  - **MouseMoved:** Quando o mouse é movido;
  - **MouseEntered/MouseExited:** Quando o ponteiro do mouse entra ou sai de um componente;



# Tratamento de Estilos com CSS em JavaFX

- **CSS (Cascading Style Sheets)** é amplamente utilizado para estilizar componentes de interface no JavaFX;
- **Com o uso de CSS, é possível modificar:**
  - Cores, fontes, margens, tamanhos de componentes;
  - Estilos dinâmicos para eventos, como o foco ou o hover;
- **Benefícios do CSS em JavaFX:**
  - Facilita a personalização da interface de forma organizada e consistente;
  - Permite separar a lógica do design, melhorando a manutenção do código;
- No JavaFX, os componentes podem ser estilizados diretamente pelo método `setStyle()`, ou, preferencialmente, através de arquivos CSS externos;
- Estilos comuns incluem:
  - **Background-color:** Cor de fundo do componente;
  - **Text-fill:** Cor do texto;
  - **Padding:** Espaçamento interno do componente;
  - **Border:** Estilo, cor e largura da borda;





# Criando Janelas Secundárias em JavaFX

- Em JavaFX, a janela principal é representada por um **Stage**;
- É possível criar janelas secundárias, chamadas de **Secondary Stages**, para exibir diferentes conteúdos ou diálogos;
- Cada janela secundária é uma nova instância da classe Stage;
- **Para criar uma nova janela:**
  - Crie uma nova instância de Stage;
  - Defina o conteúdo usando uma Scene, semelhante ao Stage principal;
  - Configure a janela (título, dimensões, etc.);
  - Use o método `show()` para exibir a nova janela;



# Diálogos de Alerta em JavaFX

- **Alert** é uma classe em JavaFX que permite exibir caixas de diálogo com mensagens para o usuário.
- Os diálogos de alerta são usados para:
  - Informar o usuário sobre uma ação;
  - Alertar sobre erros, advertências ou pedir confirmação;
- Os tipos de alerta são configurados usando a enumeração `Alert.AlertType`:
  - **INFORMATION:** Exibe informações gerais para o usuário;
  - **WARNING:** Exibe um aviso sobre uma situação que requer atenção;
  - **ERROR:** Indica um erro que ocorreu na aplicação;
  - **CONFIRMATION:** Solicita uma confirmação do usuário antes de prosseguir com uma ação;





# Exibindo Imagens no JavaFX

- **ImageView** é uma classe usada para exibir imagens em JavaFX;
- Para carregar uma imagem:
  - Crie uma instância da classe Image fornecendo o caminho da imagem;
  - Passe a instância de Image para o ImageView para exibi-la;
- É possível ajustar o tamanho da imagem usando os métodos:
  - **setFitWidth()**: Define a largura desejada da imagem;
  - **setFitHeight()**: Define a altura desejada da imagem;
- Para manter a proporção da imagem ao redimensionar:
  - Use **setPreserveRatio(true)**;
  - A imagem pode ser posicionada dentro do layout como qualquer outro componente JavaFX, como dentro de um **VBox**, **HBox**, etc;



# Desenhando Formas Geométricas em JavaFX

- JavaFX oferece classes prontas para criar formas geométricas, como:
  - **Rectangle**: Para criar retângulos;
  - **Circle**: Para criar círculos;
  - **Ellipse**, **Line**, **Polygon**, etc;
- As formas podem ser facilmente manipuladas com métodos para ajustar tamanho, posição, bordas e preenchimento;
- É possível personalizar as formas com cores e bordas, usando:
  - **setFill()**: Define a cor de preenchimento da forma;
  - **setStroke()**: Define a cor da borda;
  - **setStrokeWidth()**: Define a largura da borda;





# Canvas e Gráficos 2D em JavaFX

- **Canvas** é um componente em JavaFX usado para desenhar gráficos 2D diretamente;
- Ele fornece uma área na qual podemos **desenhar formas, imagens e texto** usando uma API de gráficos;
- Para desenhar no Canvas, utilizamos o **GraphicsContext**, que fornece métodos para desenhar linhas, formas, e aplicar cores;
- **O objeto GraphicsContext permite:**
  - Desenhar linhas, círculos, retângulos, texto, etc;
  - Configurar o estilo de preenchimento, cor, e espessura de traço;
- **Métodos comuns para desenhar:**
  - `strokeLine()`: Desenha uma linha;
  - `fillRect()`: Preenche um retângulo;
  - `strokeOval()`: Desenha um oval/círculo sem preenchimento;
  - `fillText()`: Escreve texto;



# Animações no JavaFX

- JavaFX oferece suporte robusto para **animações**, permitindo criar interações dinâmicas na interface do usuário;
- **As animações podem ser usadas para:**
  - Mover objetos na tela;
  - Alterar propriedades como opacidade, rotação, escala, etc;
  - Tornar a interface mais interativa e visualmente atrativa;
- **Timeline** é uma das classes principais para criar animações simples em JavaFX;
- Funciona ao modificar uma ou mais propriedades de um nó (componente) ao longo do tempo;
- Um **KeyFrame** representa um ponto específico no tempo, onde propriedades do nó podem ser modificadas;





# Organizando o Código JavaFX: Padrão MVC

- **MVC (Model-View-Controller)** é um padrão de arquitetura de software que separa a lógica de negócios, os dados e a interface do usuário:
  - **Model:** Representa os dados e a lógica de negócios. Gerencia o estado da aplicação;
  - **View:** Responsável pela interface do usuário. Exibe os dados e atualiza as informações conforme o estado do modelo;
  - **Controller:** Coordena a interação entre o Model e a View. Recebe ações do usuário e solicita mudanças no modelo;
- Uma boa ordem de criação pode ser: Model => Controller => fxml => Classe principal;





# JavaFx

## Conclusão