Módulo 1
Fundamentos da Linguagem Java e Desenvolvimento Orientado a Objetos

Parte 1: Fundamentos da Linguagem

1.1. Tipos de Dados Primitivos e Wrappers

- **A Explicação Concisa:** Primitivos (`int`, `double`, `boolean`) são os valores puros e básicos. São extremamente rápidos e eficientes. Wrappers (`Integer`, `Double`, `Boolean`) são objetos que "embrulham" esses valores, concedendo-lhes "superpoderes", como a capacidade de ser nulo (`null`) ou de serem usados em estruturas de dados avançadas.
- **Analogia Simples:** Um primitivo é um copo d'água (simples, direto). Um Wrapper é uma garrafa térmica (contém a água, mas oferece funcionalidades extras, como manter a temperatura).
- **Causa e Efeito:** A **causa** da existência de ambos é a necessidade de equilibrar **performance** (primitivos) e **flexibilidade** (wrappers). O **efeito** é que você usa primitivos para cálculos e laços, e wrappers para Coleções (`List<Integer>`) ou quando um valor pode estar ausente.
- **Benefício Prático:** Escolher o tipo certo otimiza seu código. Use primitivos por padrão; use wrappers quando as funcionalidades de objeto forem indispensáveis.

1.2. Operadores

- **A Explicação Concisa:** Símbolos (`+`, `==`, `&&`) que executam ações sobre seus dados, permitindo cálculos, comparações e lógica. São os "verbos" da programação.
- **Analogia Simples:** Ferramentas numa caixa: **aritméticos** (`+`, `-`) para construir; **relacionais** (`>`, `==`) para medir e comparar; **lógicos** (`&&`, `||`) para tomar decisões.
- **Causa e Efeito:** A **causa** é a necessidade de manipular e avaliar dados. O **efeito** é a capacidade de criar qualquer lógica de negócio, desde somar um carrinho de compras até validar uma senha.
- **Benefício Prático:** Sem operadores, um programa seria apenas uma lista de dados inertes. Eles dão vida e inteligência ao código.

1.3. Estruturas de Controle de Fluxo

- **A Explicação Concisa:** Comandos que direcionam o caminho de execução do código, permitindo criar bifurcações (`if`/`else`, `switch`) e repetições/loops (`for`, `while`).
- **Analogia Simples:** Um GPS. `if/else` são as bifurcações ("se houver trânsito, vire à direita"). `for/while` são as instruções de rota contínuas ("enquanto não chegar ao destino, siga em frente").

- **Causa e Efeito:** A **causa** é a necessidade de programas que reajam a diferentes cenários e evitem repetição de código. O **efeito** é um software dinâmico e eficiente.
- **Benefício Prático:** Permitem tudo, desde validar um login (`if`) até processar uma lista de usuários (`for`).

1.4. Arrays e Strings

- **A Explicação Concisa:** Um **Array** é uma coleção de tamanho fixo para itens do mesmo tipo. Uma **String** é um objeto para manipular texto, com a característica crucial de ser **imutável** (não pode ser alterada após a criação).
- **Analogia Simples:** Um **Array** é uma cartela de ovos (tamanho fixo, só ovos). Uma **String** é uma frase gravada em pedra (para mudar, você precisa de uma nova pedra). **StringBuilder/StringBuffer** são lousas brancas, onde você pode apagar e reescrever o texto livremente.
- **Causa e Efeito:** A **causa** da imutabilidade da String é a segurança e o cache. O **efeito** é que modificações repetitivas em Strings são ineficientes. Para isso, a **solução** é usar `StringBuilder` (mais rápido) ou `StringBuffer` (seguro para múltiplas threads).
- **Benefício Prático:** Arrays para coleções simples e de tamanho conhecido. `String` para todo texto. `StringBuilder` para quando você precisa construir uma String em etapas (ex: montar um longo relatório).

1.5. Métodos

- **A Explicação Concisa:** Um bloco de código reutilizável que realiza uma tarefa específica. Ajuda a organizar o código e a evitar repetição.
- **Analogia Simples:** Um liquidificador. Você passa ingredientes (**parâmetros**), ele executa sua função (**corpo do método**) e te entrega uma vitamina (**retorno**). **Sobrecarga** é ter botões diferentes no mesmo liquidificador para "bater fruta" ou "bater fruta com gelo".
- **Causa e Efeito:** A **causa** é o princípio DRY (Don't Repeat Yourself - Não se Repita). O **efeito** é um código limpo, organizado e fácil de manter.
- **Benefício Prático:** Se você precisa calcular o frete em várias partes do sistema, cria um único método `calcularFrete()` e o chama onde for necessário.

1.6. Tratamento de Exceções

- **A Explicação Concisa:** O plano de contingência do seu programa. É o mecanismo (`try-catch-finally`) para lidar com erros inesperados sem que o programa quebre.
- **Analogia Simples:** Um sistema de alarme de incêndio. `try` é a operação normal; `catch` são os sprinklers que ativam para um problema

específico; `finally` é a inspeção de segurança que ocorre no final, com ou sem incêndio.
- **Causa e Efeito:** A **causa** é que o mundo real é imprevisível (redes caem, arquivos não existem). O **efeito** é um software robusto que não trava na cara do usuário.
- **Benefício Prático:** Em vez de uma tela de erro vermelha, você pode mostrar uma mensagem amigável: "Não foi possível carregar os dados. Verifique sua conexão."

1.7. Modificadores de Acesso e Não Acesso

- **A Explicação Concisa:** Palavras-chave que definem a "visibilidade" (`public`, `private`, `protected`) e o "comportamento" (`static`, `final`) de classes, métodos e variáveis.
- **Analogia Simples:** Portas e placas em um prédio. `public` é a porta da rua; `private` é um diário trancado; `static` é o relógio na parede da recepção (pertence ao prédio, não a um morador); `final` é a placa com o número do prédio (não pode ser mudado).
- **Causa e Efeito:** A **causa** é o princípio do Encapsulamento. O **efeito** é um código seguro e organizado, onde uma parte do sistema não consegue interferir indevidamente em outra.
- **Benefício Prático:** Impede o uso incorreto de componentes, forçando a interação através de "portas" designadas e seguras.

1.8. Pacotes e Importação

- **A Explicação Concisa:** Pacotes (`package`) são pastas para organizar suas classes. `import` é o comando para "trazer" uma classe de um pacote para dentro do seu arquivo atual para poder usá-la.
- **Analogia Simples:** Um sistema de arquivos. `package` é a etiqueta na gaveta; `import` é pegar um arquivo de uma gaveta para usar.
- **Causa e Efeito:** A **causa** é a necessidade de organizar projetos grandes e evitar conflitos de nomes. O **efeito** é um projeto modular e compreensível.
- **Benefício Prático:** É impossível construir qualquer aplicação Java não trivial sem pacotes.

1.9. Conceitos Básicos de Garbage Collection (GC)

- **A Explicação Concisa:** O serviço de limpeza automático do Java. Ele encontra objetos na memória que não são mais necessários e os descarta, liberando espaço.
- **Analogia Simples:** Um robô aspirador de pó autônomo. Ele varre a casa (memória), identifica o lixo (objetos sem referência) e o recolhe, sem que você precise mandar.

- **Causa e Efeito:** A **causa** é simplificar o gerenciamento de memória para o programador. O **efeito** é a prevenção de "vazamentos de memória", uma classe comum e perigosa de bugs.
- **Benefício Prático:** Você pode focar na lógica de negócio, não na microgestão de alocação e liberação de memória.

Parte 2: Estruturas de Dados e Coleções (Collections Framework)

Se os tipos primitivos são os tijolos, as coleções são as diferentes formas de construir paredes e salas para organizar esses tijolos.

2.1. Interfaces Principais e Implementações

- **A Explicação Concisa:** O Java fornece "plantas" (Interfaces) para diferentes tipos de coleções de dados, e "construções" concretas (Implementações) baseadas nessas plantas.
- **Analogia Simples:** Pense em veículos. `List`, `Set` e `Map` são categorias de veículos (a interface).
    - **`List` (Lista):** Um trem. Os itens (`passageiros`) estão em ordem e podem haver duplicatas.
        - `ArrayList`: Um trem-bala. Ótimo para ir direto a um vagão específico (acesso por índice: `get(i)`).
        - `LinkedList`: Um trem de carga. Ótimo para engatar e desengatar vagões no meio (inserção/remoção).
    - **`Set` (Conjunto):** Uma festa de convidados VIP. Não permite duplicatas. Se o "João" já está na festa, não dá pra adicionar outro "João".
        - `HashSet`: A festa mais rápida. Não se importa com a ordem de chegada, só quer garantir a unicidade rapidamente.
        - `LinkedHashSet`: Uma festa organizada. Mantém a ordem em que os convidados chegaram.
        - `TreeSet`: Uma festa chique. Organiza os convidados por ordem alfabética (ou outra ordem natural).
    - **`Map` (Mapa):** Um dicionário. Cada palavra (**chave**, que é única) aponta para sua definição (**valor**).
        - `HashMap`: O dicionário mais rápido. Não se importa com a ordem das palavras.
        - `LinkedHashMap`: Mantém a ordem em que você inseriu as palavras.
        - `TreeMap`: Mantém as palavras em ordem alfabética.
    - **`Queue` / `Deque` (Fila):** Uma fila de banco. `Queue` é a fila normal (primeiro a entrar, primeiro a sair). `Deque` é uma fila "dupla", onde se pode entrar e sair pelas duas pontas.

        - `PriorityQueue`: Uma fila de emergência de hospital. A ordem de saída é baseada na prioridade (gravidade do paciente), não na ordem de chegada.
        - `ArrayDeque`: A implementação mais comum e eficiente para filas e pilhas (último a entrar, primeiro a sair).
- **Causa e Efeito:** A **causa** de tantas opções é que não existe uma "coleção perfeita". Cada uma resolve um problema específico de performance. O **efeito** é que a escolha correta da coleção pode impactar dramaticamente a velocidade e o consumo de memória da sua aplicação.
- **Benefício Prático:** Escolha `ArrayList` como padrão para listas. Use `LinkedList` se você remove/adiciona muitos itens no meio da lista. Use `HashSet` para garantir unicidade rapidamente. Use `HashMap` sempre que precisar de uma associação chave-valor.

2.2. Iteradores e Generics

- **Iterator:** É um objeto que permite percorrer uma coleção item por item de forma segura, principalmente para remover elementos durante a iteração. É como um "dedo" que aponta para cada item da lista, um de cada vez.
- **Generics (`< >`):** É o mecanismo de segurança que permite especificar o tipo de dado que uma coleção pode conter (ex: `List<String>`). Isso garante que você só poderá adicionar `Strings` àquela lista, e o compilador te avisará se você tentar adicionar um número. É como etiquetar a cartela de ovos para garantir que ninguém coloque uma maçã lá dentro. O benefício é a segurança de tipo em tempo de compilação, evitando erros em tempo de execução.

Parte 3: Programação Orientada a Objetos (POO)

POO não é sobre código, é sobre como pensar e estruturar o código. É modelar o mundo real em termos de "objetos" que têm características (atributos) e comportamentos (métodos).

3.1. Pilares da POO

- **Analogia Geral:** Construir um carro usando componentes.
    1. **Encapsulamento:** O motor do carro. Você, como motorista, não precisa saber como os pistões e as válvulas funcionam. Você só interage com a "interface" (a chave de ignição e o acelerador). O mecanismo interno está protegido e escondido.
    2. **Herança:** A relação entre "Carro" e "Carro Esportivo". Um Carro Esportivo **é um** Carro, então ele **herda** características básicas como ter rodas e um motor, mas adiciona suas próprias, como um aerofólio e um motor turbo. Permite o reuso de código.

3. **Polimorfismo:** "Muitas formas". O ato de "ligar" um carro. Ligar um carro elétrico é diferente de ligar um carro a combustão, mas o conceito (o método `ligar()`) é o mesmo para o motorista. Permite que objetos diferentes respondam à mesma mensagem de maneiras diferentes.
4. **Abstração:** O painel do carro. Ele te mostra apenas a informação essencial (velocidade, combustível) e esconde toda a complexidade dos sensores e da eletrônica que geram essa informação. É focar no "o que" um objeto faz, e não no "como" ele faz.

- **Benefício Prático:** POO resulta em código mais organizado, reutilizável, flexível e fácil de manter, especialmente em sistemas grandes e complexos.

3.2. Classes, Objetos, Interfaces e Mais

- **Classe:** A planta ou o projeto de um objeto. Ex: A planta de uma "Casa".
- **Objeto:** A instância concreta de uma classe. Ex: "a minha casa", "a sua casa".
- **Interface:** Um contrato. Define "o que" uma classe deve fazer, mas não "como". Uma classe pode `implementar` várias interfaces. Ex: A interface `Voavel` define que deve haver um método `voar()`, e tanto `Aviao` quanto `Passaro` podem implementá-la, cada um à sua maneira.
- **Classe Abstrata:** Uma classe "incompleta" que mistura o concreto (métodos já prontos) e o abstrato (métodos que as classes filhas devem implementar). Serve como uma base comum para uma família de classes.
- **Enumeração (`enum`):** Uma forma de criar um tipo com um conjunto fixo de constantes. Ex: `enum DiaDaSemana { SEGUNDA, TERCA, ... }`. É muito mais seguro e legível do que usar números ou Strings.

3.3. Composição e Agregação

São duas formas de um objeto ser formado por outros.

- **Composição (Relação "tem-um" forte):** O objeto "parte" não existe sem o "todo". Um `Motor` faz parte de um `Carro`. Se o carro é destruído, o motor vai junto. É uma relação de posse.
- **Agregação (Relação "tem-um" fraca):** O objeto "parte" pode existir independentemente. Um `Carro` tem um `PlayerDeMusica`. Se o carro for destruído, o Player de Música pode ser retirado e usado em outro lugar.

3.4. Princípios SOLID

São cinco princípios de design que tornam o software mais robusto e manutenível.

- **S - Single Responsibility Principle (Responsabilidade Única):** Uma classe deve ter apenas um motivo para mudar. (Uma classe `Calculadora` calcula, uma classe `ImpressoraDeRelatorio` imprime).
- **O - Open/Closed Principle (Aberto/Fechado):** Aberto para extensão, fechado para modificação. (Para adicionar um novo tipo de pagamento, você cria uma nova classe, não altera a classe existente que processa pagamentos).
- **L - Liskov Substitution Principle (Substituição de Liskov):** Classes filhas devem ser substituíveis por suas classes mãe sem quebrar o sistema. (Se você tem um método que funciona com um `Passaro`, ele também deve funcionar com um `Pato`, que é um tipo de `Passaro`).
- **I - Interface Segregation Principle (Segregação de Interfaces):** É melhor ter várias interfaces pequenas e específicas do que uma grande e genérica. (Não force uma classe `ImpressoraSimples` a implementar métodos como `escanear()` ou `enviarFax()`).
- **D - Dependency Inversion Principle (Inversão de Dependência):** Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. (Sua classe `GeradorDeNotaFiscal` não deve depender diretamente da classe `BancoDeDadosOracle`, mas sim de uma interface `RepositorioDeNotas`).

3.5. Design Patterns (Padrões de Projeto)

- **A Explicação Concisa:** Soluções testadas e comprovadas para problemas recorrentes no design de software.
- **Analogia Simples:** Receitas de culinária. Se você quer fazer um bolo, não precisa reinventar a roda, você segue uma receita que já funciona.
- **Exemplos Comuns:**
    - **Singleton:** Garante que uma classe tenha apenas uma única instância em todo o programa. (Ex: A classe que gerencia a conexão com o banco de dados).
    - **Factory (Fábrica):** Cria objetos sem expor a lógica de criação ao cliente. (Você pede uma "pizza" à fábrica, e ela te entrega uma `PizzaCalabresa` ou `PizzaMussarela` pronta, sem que você precise saber como cada uma é feita).
    - **Strategy (Estratégia):** Permite que você escolha um algoritmo em tempo de execução. (Em um e-commerce, você pode selecionar a `EstrategiaDeCalculoDeFrete` entre "Sedex", "PAC" ou "Transportadora").
    - **Observer (Observador):** Permite que um objeto (o "observado") notifique uma lista de outros objetos (os "observadores") sobre qualquer mudança de estado. (Quando a `Revista` (observado) lança uma nova edição, todos os `Assinantes` (observadores) são notificados).

Parte 4: Multithreading e Concorrência

É a arte de fazer seu programa executar múltiplas tarefas ao mesmo tempo, melhorando a performance e a responsividade.

4.1. Threads, Processos e Criação

- **A Explicação Concisa:** Um **processo** é um programa em execução (como o seu navegador). Uma **thread** é uma linha de execução dentro desse processo. Um processo pode ter várias threads rodando "simultaneamente".
- **Analogia Simples:** Um restaurante é o **processo**. Os cozinheiros são as **threads**. Vários cozinheiros podem trabalhar ao mesmo tempo em diferentes partes de um pedido para que ele saia mais rápido.
- **Criação:** Você pode criar threads estendendo a classe `Thread` ou, de forma mais comum e flexível, implementando a interface `Runnable`.

4.2. Sincronização e Deadlock

- **Sincronização (`synchronized`):** O mecanismo para garantir que apenas uma thread possa acessar um recurso compartilhado por vez, evitando inconsistências. Na analogia do restaurante, é a regra que diz: "Apenas um cozinheiro pode usar a fritadeira por vez".
- **Deadlock e Livelock:** São condições de erro em concorrência.
    - **Deadlock (Impasse):** Dois cozinheiros parados, um esperando a panela que o outro tem, e o outro esperando a faca que o primeiro tem. Nenhum dos dois pode continuar.
    - **Livelock:** Dois cozinheiros tentam passar um pelo outro em um corredor estreito. Ambos dão um passo para o lado, se bloqueiam de novo, dão um passo para o outro lado, e ficam nesse "loop educado" sem nunca conseguir passar.

4.3. Conceitos e Ferramentas (`java.util.concurrent`)

- **Atomicidade:** Uma operação que acontece por inteiro ou não acontece. É indivisível.
- **Visibilidade:** Garantir que as mudanças feitas por uma thread em um dado compartilhado sejam visíveis para as outras threads.
- **Ordenação:** Garantir que as instruções sejam executadas na ordem que você espera.
- **`java.util.concurrent`:** Um pacote poderoso com ferramentas de alto nível para gerenciar concorrência de forma mais fácil e segura que o `synchronized` básico.
    - **`ExecutorService`:** Gerencia um "pool" de threads (um grupo de cozinheiros prontos para trabalhar), para que você não precise criar e destruir threads toda hora.

    - **`Semaphore`:** Um semáforo que limita o número de threads que podem acessar um recurso. (Ex: "Apenas 5 cozinheiros podem usar o balcão de preparo ao mesmo tempo").

Parte 5: Input/Output (I/O)

Como seu programa lê e escreve dados de/para fontes externas como arquivos, rede, etc.

- **A Explicação Concisa:** Java usa o conceito de **Streams** (fluxos) para lidar com I/O. Você abre um fluxo de uma fonte (um arquivo) e lê ou escreve dados de forma sequencial.
- **Analogia Simples:** Um sistema de canos de água.
    - **`InputStream` / `Reader`:** Canos que trazem água para dentro (leitura).
    - **`OutputStream` / `Writer`:** Canos que levam água para fora (escrita).
    - **Byte Streams (`InputStream`/`OutputStream`):** Canos que transportam a água pura, molécula por molécula (dados brutos, como imagens ou áudio).
    - **Character Streams (`Reader`/`Writer`):** Canos que transportam suco (texto), já com uma codificação (`UTF-8`) para interpretar os caracteres corretamente.
- **`BufferedReader`, `PrintWriter`:** São "envoltórios" que adicionam um "reservatório" (buffer) ao cano. Isso é muito mais eficiente, pois em vez de ler/escrever uma gota por vez, você enche o reservatório e descarrega tudo de uma vez.
- **Serialização:** O processo de converter um objeto Java em um fluxo de bytes para que ele possa ser salvo em um arquivo ou enviado pela rede e, posteriormente, reconstruído.

Parte 6: Java 17 e Novidades

Recursos modernos que tornam o Java mais conciso, expressivo e seguro.

6.1. Lambdas e Streams API

- **A Explicação Concisa:** Uma maneira funcional e declarativa de processar coleções de dados.
- **Analogia Simples:** Uma linha de montagem industrial. Em vez de um artesão (`for` loop) fazer todo o processo em um item de cada vez, você coloca todos os itens em uma esteira (**`Stream`**) e eles passam por estações de trabalho especializadas (**`filter`**, **`map`**, **`collect`**).
- **Benefício Prático:** Código muito mais limpo e legível para manipulação de coleções. Compare:

    - **Antes:** Várias linhas com `for` e `if` para filtrar e transformar uma lista.
    - **Agora:** Uma única linha fluente de código.
- **Interfaces Funcionais e Method References: Interface Funcional** é uma interface com um único método abstrato, o "alvo" de uma lambda. **Method Reference** (`::`) é um atalho para uma lambda que apenas chama um método existente.

6.2. Optional, Records, Sealed Classes

- **`Optional<T>`:** Um "contêiner" que pode ou não ter um valor dentro. É uma forma explícita e segura de lidar com a possibilidade de valores nulos, evitando `NullPointerException`. Em vez de perguntar `if (valor ≠ null)`, você usa métodos como `.ifPresent()` ou `.orElse()`.
- **Record Classes (`record`):** Uma forma super concisa de criar classes que são apenas "carregadoras de dados" imutáveis (como um DTO). O compilador gera automaticamente construtores, `getters`, `equals()`, `hashCode()` e `toString()`. Reduz drasticamente o código repetitivo.
- **Sealed Classes (`sealed`):** Permitem que você restrinja quais outras classes podem estender ou implementar sua classe/interface. Dá a você controle total sobre sua hierarquia de herança, tornando o código mais seguro e o `switch` mais poderoso.

Parte 7: Testes Unitários (Conceitos Básicos)

A prática de escrever código para testar seu próprio código, garantindo que cada pequena "unidade" (método) funcione como esperado.

- **A Explicação Concisa:** Testes unitários validam as menores partes do seu código de forma isolada para garantir que elas estão corretas.
- **Analogia Simples:** Um inspetor de qualidade em uma fábrica de carros. Antes de montar o carro inteiro, ele testa cada peça individualmente: o parafuso aguenta a pressão? O pneu não está furado? O farol acende?
- **JUnit:** A "bancada de testes" (framework) mais popular para Java.
- **Anotações Básicas:**
    - `@Test`: Marca um método como um caso de teste.
    - `@BeforeEach` / `@AfterEach`: Métodos que rodam antes/depois de **cada** teste (ex: para criar e limpar um objeto).
    - `@BeforeAll` / `@AfterAll`: Métodos que rodam uma única vez antes/depois de **todos** os testes da classe (ex: para iniciar uma conexão com um banco de dados de teste).
    - **`Assertions`:** São os comandos que verificam o resultado. `assertEquals(esperado, real)` verifica se dois valores são iguais. Se a verificação falhar, o teste falha.
- **Benefício Prático:** Aumenta a confiança para fazer alterações no código (refatoração), documenta o comportamento esperado de um método

e ajuda a encontrar bugs muito mais cedo no processo de desenvolvimento.