Claro! Aqui está uma versão melhorada, com explicações mais claras, linguagem mais objetiva e técnica, mantendo o tom didático:

2.2. Encapsulamento

2.2.1. Definição e Exemplo de Encapsulamento

No exercício anterior, a implementação da classe `Conta` funciona, mas apresenta um ponto que pode (e deve) ser melhorado para refletir melhor as boas práticas da programação orientada a objetos. Observe o seguinte trecho:

```
conta.saldo = 1000000000000;
```

Se o atributo `saldo` da classe `Conta` estiver declarado como `public`, qualquer parte do código pode acessá-lo diretamente e alterá-lo para qualquer valor — inclusive valores absurdos, como no exemplo acima. Isso representa um risco enorme, especialmente em um sistema bancário, pois permite que o estado interno de um objeto seja manipulado livremente, quebrando a integridade dos dados.

Esse cenário viola um dos princípios fundamentais da programação orientada a objetos: **o encapsulamento**.

> **Encapsulamento** é o princípio que determina que os detalhes internos de uma classe devem ser protegidos e não expostos diretamente ao mundo exterior. A interação com esses dados deve ocorrer por meio de métodos controlados, chamados de **métodos de acesso** (getters e setters), ou por operações seguras fornecidas pela própria classe.

No exemplo bancário, permitir a alteração direta do saldo compromete toda a lógica do sistema. Em uma aplicação real, o saldo de uma conta só pode ser alterado por operações válidas e registradas, como **depósitos** e **saques**, garantindo segurança e rastreabilidade.

2.2.2. Modificadores de Visibilidade: `private`, `public` e `protected`

Para aplicar o encapsulamento corretamente, usamos **modificadores de visibilidade**. Eles controlam o nível de acesso de campos e métodos de uma classe. Em Java, os principais são:

Modificador	Acesso permitido por
`public`	Qualquer classe

`private`	Apenas na própria classe
`protected`	Na própria classe e em subclasses (inclusive em pacotes diferentes)
sem modificador (default)	Classes no **mesmo pacote**

> ❗ **Importante**: Evite declarar atributos de uma classe como `public`. Isso expõe diretamente os dados e compromete a integridade da lógica interna. O ideal é utilizar `private` e fornecer acesso controlado por métodos.

Vamos observar um exemplo:

Classe Cliente (com campos privados)

```
package banco.modelo;

public class Cliente {

    private int codigo;
    private String nome;
    private String cidade;
    private String estado;

    private static int quantidade;

    // Métodos getters e setters serão adicionados depois
}
```

Ao tornar os campos privados, qualquer tentativa de acessá-los diretamente fora da classe resultará em erro de compilação, como no exemplo abaixo:

Classe Principal (com erros após encapsulamento)

```
Cliente cliente = new Cliente();
cliente.nome = JOptionPane.showInputDialog(null, "Nome do Cliente: "); // Erro
cliente.cidade = JOptionPane.showInputDialog(null, "Cidade do Cliente: "); // Erro
cliente.estado = JOptionPane.showInputDialog(null, "Estado do Cliente: "); // Erro
```

Esses erros acontecem porque os campos agora são `private`. Para contornar isso de maneira correta, utilizaremos os métodos **getter** (para obter o valor de um campo) e **setter** (para modificar o valor de um campo). Esses métodos nos permitem controlar o acesso e até adicionar regras de validação, se necessário.

✅ Resumo

- **Encapsulamento** protege os dados internos de uma classe, impedindo acessos e modificações diretas e não autorizadas.
- Campos devem ser declarados como `private` para proteger sua integridade.
- O acesso a esses campos deve ser feito através de **getters e setters**.
- Usar `public` diretamente nos atributos compromete a segurança, manutenibilidade e consistência do código.

Claro! Aqui está a versão **melhorada, mais didática e com explicações mais claras e objetivas** do conteúdo sobre encapsulamento com **getters e setters**:

2.2. Encapsulamento

2.2.3. Usando Getters e Setters

Em Java (e na maioria das linguagens orientadas a objetos), é **boa prática proteger os dados de uma classe** contra alterações diretas por outras classes. Para isso, usamos o **encapsulamento**.

Encapsulamento significa **ocultar os detalhes internos de uma classe**, permitindo que seu uso seja feito de forma controlada, sem comprometer sua integridade. Isso é feito usando **modificadores de acesso** (como `private`) e os chamados **métodos getters e setters**.

🧩 O que são Getters e Setters?

- **Getter**: Método que permite **ler** o valor de um atributo `private`.
- **Setter**: Método que permite **modificar** o valor de um atributo `private`, de forma controlada.

✅ Exemplo de Getter

Imagine que temos um campo `nome`, declarado como `private` dentro da classe `Cliente`. Para acessar seu valor de fora da classe, criamos um método `getNome()`:

public String getNome() {

```
    return nome;
}
```

Agora, ao invés de acessar diretamente `cliente.nome`, usamos:

```
String nomeDoCliente = cliente.getNome();
```

Isso permite que **o controle de acesso continue com a classe**, e **não diretamente com o usuário da classe**.

🧠 Mas qual a vantagem disso?

Com o getter, **podemos manipular ou formatar o valor antes de retorná-lo**. Por exemplo, imagine que queremos retornar apenas o **primeiro nome** do cliente:

```
public String getNome() {
    return (nome.contains(" ") ? nome.substring(0, nome.indexOf(' ')) : nome);
}
```

Assim, se o nome for "João Silva", o método retornará apenas "João".

✅ Exemplo de Setter

Para **modificar** o valor de um atributo `private`, usamos um método `setNome()`:

```
public void setNome(String nome) {
    this.nome = nome;
}
```

Usamos `this.nome` para indicar que estamos atribuindo valor ao atributo da classe, e não a uma variável local.

Agora, ao invés de fazer:

cliente.nome = "Maria"; // ERRADO - campo private

Fazemos:

cliente.setNome("Maria"); // CERTO

🚧 Controle de entrada com Setter

Com o setter, podemos validar os dados antes de alterar o atributo. Exemplo:

```
public void setNome(String nome) {
    if (nome.isEmpty()) {
        this.nome = "NÃO FORNECIDO";
    } else {
        this.nome = nome;
    }
}
```

Dessa forma, impedimos que o nome seja salvo vazio.

💡 Aplicando na prática

Vamos atualizar o código da classe `Cliente` com um getter e setter para `nome`:

```
public class Cliente {
    private String nome;

    public String getNome() {
        return (nome.contains(" ") ? nome.substring(0, nome.indexOf(' ')) : nome);
    }
```

```
    public void setNome(String nome) {
        if (nome.isEmpty()) {
            this.nome = "NÃO FORNECIDO";
        } else {
            this.nome = nome;
        }
    }
}
```

🔧 Usando na classe Principal

Antes (errado):

cliente.nome = JOptionPane.showInputDialog(null, "Nome do Cliente: ");

Depois (certo):

cliente.setNome(JOptionPane.showInputDialog(null, "Nome do Cliente: "));

📝 Resumo

Elemento	Função
`private`	Restringe acesso direto ao campo
`getter`	Permite **ler** um valor com controle

`setter`	Permite **modificar** um valor com validação
`this`	Refere-se ao campo da própria classe

Se quiser, posso montar um **exemplo completo com `Cliente`, `Principal` e uso dos getters/setters** para você copiar e testar. Quer?