FIAP - TDS - Prof. Tiago Ferrer

Domain Driven Design

Semana 3 - 11/03/24 : 15/03/2024

Objetivos

Classes, Atributos e Métodos

As classes são os blocos de construção fundamentais na programação baseada em Orientação a Objetos (OO). Fazendo uma analogia com a construção civil, podemos imaginar uma classe como uma planta baixa de uma casa. A planta define a estrutura da casa, incluindo o número de cômodos, portas e janelas. Analogamente, uma classe define a estrutura de um objeto, especificando suas propriedades (atributos) e comportamentos (métodos).

Por exemplo, podemos ter uma classe Carro que define as propriedades de um carro, como cor, modelo e ano. A classe também pode conter métodos que simulam o comportamento de um carro, como `acelerar()`, `frear()` e `buzinar()`.

Veja alguns pontos-chave sobre classes em Java:

- **Atributos ou Campos (*Fields*)**: Representam as características de um objeto, como seus dados.
- **Métodos**: Definem o comportamento de um objeto, ditando as ações que ele pode realizar.
- **Objetos**: São instâncias criadas a partir de uma classe. Pense em um objeto como uma casa construída a partir da planta baixa. Vários objetos podem ser criados a partir da mesma classe, assim como podemos construir várias casas a partir da mesma planta.

Resumindo, classes servem como modelos para criar objetos, encapsulando suas propriedades e comportamentos. Isso promove a organização, reusabilidade e modularidade do código Java.

```
public class Carro {

    private int velocidade;

    private String cor;

    private int portas;

    private int ano;

    private String modelo;

    public void acelerar(){
        velocidade++;
    }

    public void frear(){
        if(velocidade > 0){
            velocidade--;
        }
    }
}
```

Visibilidade de Métodos, Atributos e Classes em Java: Controlando o Acesso em Seu Código

A visibilidade em Java é um conceito fundamental que controla o acesso a métodos, atributos e classes. Ela define quem pode acessar e modificar esses elementos do código, garantindo a segurança, modularidade e flexibilidade do seu software.

Níveis de Visibilidade:

- **Public**: Permite acesso a partir de qualquer lugar no código.
- **Protected**: Permite acesso apenas dentro da classe e suas subclasses.
- **Package-private**: Permite acesso apenas dentro do mesmo pacote.
- **Private**: Permite acesso apenas dentro da própria classe.

Visibilidade de Métodos:

- **Public**: Métodos públicos podem ser invocados por qualquer classe que tenha acesso à classe que os define.
- **Protected**: Métodos protegidos podem ser invocados apenas por subclasses da classe que os define.
- **Package-private**: Métodos package-private podem ser invocados apenas por classes dentro do mesmo pacote.
- **Private**: Métodos privados podem ser invocados apenas pela própria classe que os define.

Visibilidade de Atributos:

- **Public**: Atributos públicos podem ser acessados e modificados por qualquer classe que tenha acesso à * classe que os define.
- **Protected**: Atributos protegidos podem ser acessados e modificados apenas por subclasses da classe que os define.
- **Package-private**: Atributos package-private podem ser acessados e modificados apenas por classes dentro do mesmo pacote.
- **Private**: Atributos privados podem ser acessados e modificados apenas pela própria classe que os define.

Visibilidade de Classes:

- **Public**: Classes públicas podem ser utilizadas por qualquer classe que tenha acesso ao pacote que as define.
- **Package-private**: Classes package-private podem ser utilizadas apenas por classes dentro do mesmo pacote.

Recomendações:

- Utilize o nível de visibilidade mais restritivo possível para cada método, atributo e classe.
- Evite o uso de visibilidade pública para atributos e métodos internos da classe.
- Utilize visibilidade protected para métodos e atributos que precisam ser acessados por subclasses.
- Utilize visibilidade package-private para métodos e atributos que precisam ser acessados por classes dentro do mesmo pacote.

Exemplo:

```java
public class Pessoa {

    private String nome; // Atributo privado

    public void setNome(String nome) { // Método público
        this.nome = nome;
    }

    protected String getNome() { // Método protegido
        return nome;
    }

}
```

```java
public class Cliente extends Pessoa {

    public void mostrarNome() {
        System.out.println(getNome()); // Acessa método protegido da superclasse
    }

}
```

Neste exemplo, o atributo `nome` é privado e só pode ser acessado pelo método `setNome`. O método `getNome` é protegido e só pode ser acessado pela própria classe `Pessoa` e suas subclasses. A classe `Cliente` pode acessar o método `getNome` da superclasse `Pessoa` por ser uma subclasse.

A tabela abaixo sintetiza os acessos em relação as visibilidades:

Acesso de membros na:	private	protected	package	public
Implementação	sim	sim	sim	sim
Instâncias (no pacote)	não	sim	sim	sim
Instâncias (fora do pacote)	não	não	não	sim

A visibilidade é um recurso poderoso que permite controlar o acesso a métodos, atributos e classes em Java. Ao usar os níveis de visibilidade de forma adequada, você pode garantir a segurança, modularidade e flexibilidade do seu código, tornando-o mais fácil de entender, manter e reutilizar.

Métodos de acesso - Getters e Setters

O que são Gets e Sets?

Gets e sets são métodos especiais em Java que servem para acessar e modificar os atributos privados de uma classe.

- **Getters**: Retornam o valor de um atributo privado.
- **Setters**: Definem o valor de um atributo privado.

Necessidade de Gets e Sets:

- **Encapsulamento**: Permitem controlar o acesso aos atributos da classe, protegendo-os de alterações indevidas.
- **Validação**: Permitem realizar validações nos valores antes de serem atribuídos aos atributos.
- **Reutilização**: Permitem padronizar o acesso aos atributos, facilitando a reutilização do código.

Importância de evitar Gets e Sets desnecessários:

- **Eficiência**: A criação de métodos desnecessários pode tornar o código mais lento e menos eficiente.
- **Manutenabilidade**: Aumenta a quantidade de código que precisa ser mantida, tornando-a mais complexa.
- **Legibilidade**: Torna o código mais difícil de entender, pois aumenta o número de métodos sem necessidade.

Quando usar Gets e Sets:

- **Atributos privados**: Utilize gets e sets para acessar e modificar atributos privados.
- **Validação**: Utilize gets e sets para realizar validações nos valores antes de serem atribuídos aos atributos.
- **Reutilização**: Utilize gets e sets para padronizar o acesso aos atributos, facilitando a reutilização do código.

Dicas para evitar Gets e Sets desnecessários:

- **Métodos alternativos**: Utilize métodos específicos para realizar operações complexas nos atributos, em vez de gets e sets simples.
- **Considere o contexto**: Avalie se a necessidade de gets e sets justifica a criação de métodos adicionais.

Exemplo:

```
public class Pessoa {

    private String nome;
    private int idade;

    public String getNome() {
        return nome;
    }

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

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        if (idade < 0) {
            throw new IllegalArgumentException("Idade inválida!");
        }
        this.idade = idade;
    }

}
```

Neste exemplo, os gets e sets são necessários para proteger os atributos `nome` e `idade` de alterações indevidas. O setter de `idade` também realiza uma validação para garantir que a idade seja um valor positivo.

Classes Anêmicas: Um Problema Comum em Java e DDD

Em Java, classes anêmicas são classes que não possuem comportamento significativo. Elas servem apenas como containers de dados, armazenando atributos e *getters/setters*. Isso pode levar a diversos problemas, como:

- **Falta de coesão**: A classe não possui um único propósito claro, tornando-a difícil de entender e manter.

- **Falta de reutilização**: A classe não pode ser facilmente reutilizada em outros projetos, pois seu comportamento é específico para o contexto atual.
- **Código frágil**: Alterações nos atributos da classe podem ter um efeito cascata no código que a utiliza.

DDD e Classes Anêmicas:

O Domain-Driven Design (DDD) é uma metodologia de desenvolvimento de software que coloca o foco no **domínio do problema**. No DDD, as classes representam entidades do domínio e possuem um comportamento rico e significativo. Isso evita a criação de classes anêmicas.

Exemplos de Classes Anêmicas:

- **Classe Pessoa**: Uma classe Pessoa que apenas armazena nome, idade e sexo é anêmica. Ela não possui nenhum comportamento significativo, como calcular a idade ou verificar se a pessoa é maior de idade.
- **Classe Endereço**: Uma classe Endereço que apenas armazena logradouro, número, complemento, cidade e estado é anêmica. Ela não possui nenhum comportamento significativo, como formatar o endereço para impressão ou calcular a distância entre dois endereços.

Como evitar Classes Anêmicas:

Aplique os princípios do DDD: Crie classes que representam entidades do domínio e que possuem um comportamento rico e significativo. Utilize métodos e interfaces: Utilize métodos para encapsular o comportamento da classe e interfaces para definir contratos entre classes. Evite getters/setters triviais: Utilize getters/setters apenas quando necessário. Se um atributo não precisa ser modificado, declare-o como final. Exemplos de Classes com Comportamento:

- **Classe Pessoa**: Uma classe Pessoa que calcula a idade, verifica se a pessoa é maior de idade e formata o nome para diferentes contextos.
- **Classe Endereço**: Uma classe Endereço que formata o endereço para impressão, calcula a distância entre dois endereços e valida o CEP.

Classes anêmicas são um problema comum em Java que pode levar a diversos problemas. Ao aplicar os princípios do DDD e evitar getters/setters triviais, você pode criar classes com comportamento rico e significativo, tornando seu código mais coeso, reutilizável e robusto.

Enum em Java: Definição, Usos e Exemplos

Um **Enum** em Java é um tipo especial de classe que define um conjunto de valores constantes e relacionados. Pense em um Enum como um “menu” de opções pré-definidas para um determinado tipo de dado.

Quando usar Enum:

- Para representar um conjunto de valores finitos e relacionados, como:
  - Dias da semana (`SEGUNDA` , `TERCA` , `QUARTA` , ...)
  - Cores (`VERMELHO` , `VERDE` , `AZUL` , ...)
  - Tamanhos de roupas (`P` , `M` , `G` , ...)
- Para melhorar a legibilidade e segurança do código, evitando “strings mágicas” e valores inconsistentes.
- Para facilitar a comparação e manipulação de valores, utilizando métodos específicos de Enum.

Como usar Enum:

- **Declaração**: Utilize a palavra-chave enum seguida do nome do Enum e dos valores entre chaves.

Exemplos:

```
enum DiaDaSemana {
    SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO, DOMINGO
}

enum Cor {
    VERMELHO, VERDE, AZUL, AMARELO, PRETO, BRANCO
}

enum TamanhoRoupa {
    P, M, G, GG, XG
}
```

- **Utilização**: Crie variáveis do tipo Enum e utilize os valores como constantes.
- **Exemplos**:

```
DiaDaSemana hoje = DiaDaSemana.QUARTA;
Cor carro = Cor.VERMELHO;
TamanhoRoupa camisa = TamanhoRoupa.M;

if (hoje == DiaDaSemana.SEXTA) {
    System.out.println("Ufa, fim de semana!");
}

switch (carro) {
    case VERMELHO:
        System.out.println("Carro vermelho é paixão!");
        break;
    case VERDE:
        System.out.println("Carro verde é esperança!");
        break;
    default:
        System.out.println("Carro de outra cor!");
}
```

Vantagens de usar Enum:

- **Segurança**: Evita erros de digitação e valores inconsistentes.
- **Legibilidade**: Torna o código mais fácil de entender e manter.
- **Reutilização**: Permite a criação de tipos de dados customizados e reutilizáveis.
- **Facilidade de uso**: Oferece métodos para comparação, ordenação e outras operações.

Enum é um recurso poderoso em Java que permite definir conjuntos de valores constantes e relacionados, melhorando a legibilidade, segurança e reutilização de código. Explore os recursos avançados para criar tipos de dados customizados e ainda mais eficientes

Desafio

Desafio Java: Rede Social de Animais de Estimação Crie uma rede social para animais de estimação em Java, onde os usuários podem criar perfis para seus bichinhos, adicionar amigos, compartilhar fotos e interagir uns com os outros.

Requisitos:

- Enum: Crie um enum para representar os diferentes tipos de animais de estimação (ex: Cachorro, Gato, Coelho, etc.).
- Múltiplos objetos: Crie classes para representar os diferentes tipos de objetos na rede social, como:

    - Usuario: Nome, email, senha, foto de perfil, lista de amigos, lista de animais de estimação.
    - Animal de estimação: Tipo, nome, raça, idade, foto, lista de amigos.
    - Postagem: Texto, foto, data/hora, lista de curtidas, lista de comentários.

- Relacionamentos entre objetos:

    - Um usuário pode ter vários animais de estimação.
    - Um animal de estimação pode ter vários amigos (outros animais de estimação).
    - Um usuário pode ter vários amigos (outros usuários).
    - Um usuário pode curtir e comentar as postagens de outros usuários.

- Sem classes anêmicas: As classes devem ter atributos e métodos que representam seus comportamentos e características.

- Método main: Implemente a lógica principal da rede social no método main, incluindo:

    - Criação de alguns usuários e animais de estimação.
    - Adição de amigos para usuários e animais de estimação.
    - Criação de algumas postagens.
    - Curtida e comentário em postagens.

Recursos adicionais e de referências:

Artigo sobre classes anêmicas: https://www.alura.com.br/artigos/o-que-e-modelo-anemico-e-por-que-fugir-dele?utm_term=&utm_campaign=%5BSearch%5D+%5BPerformance%5D+-+Dynamic+Search+Ads+-+Artigos+e+Conte%C3%BAdos&utm_source=adwords&utm_medium=ppc&hsa_acc=7964138385&hsa_cam=11384329873&hsa_grp=111087461203&hsa_ad=687448474447&hsa_src=g&hsa_tgt=dsa-2276348409543&hsa_kw=&hsa_mt=&hsa_net=adwords&hsa_ver=3&gad_source=1&gclid=CjwKCAjw17qvBhBrEiwA1rU9w43obqsIk9KX1ptDq_AVOieh3o7k6lVfGkhpo7J44m3QqTVMTkSUKBoCjS0QAvD_BwE (https://www.alura.com.br/artigos/o-que-e-modelo-anemico-e-por-que-fugir-dele?utm_term=&utm_campaign=%5BSearch%5D+%5BPerformance%5D+-+Dynamic+Search+Ads+-+Artigos+e+Conte%C3%BAdos&utm_source=adwords&utm_medium=ppc&hsa_acc=7964138385&hsa_cam=11384329873&hsa_grp=111087461203&hsa_ad=687448474447&hsa_src=g&hsa_tgt=dsa-2276348409543&hsa_kw=&hsa_mt=&hsa_net=adwords&hsa_ver=3&gad_source=1&gclid=CjwKCAjw17qvBhBrEiwA1rU9w43obqsIk9KX1ptDq_AVOieh3o7k6lVfGkhpo7J44m3QqTVMTkSUKBoCjS0QAvD_BwE)

VERNON,Vaugh. Implementando Domain-Driven Design, ed. São Paulo: Alta Books, 2016.

JANDLR JR., Peter. Java Guia do Programador, 4. ed. São Paulo: Novatec Editora, 2021.