FIAP - TDS - Prof. Tiago Ferrer

Domain Driven Design

Semana 5 - 25/03/24 : 29/03/2024

Construtores em Java

Em Java, os construtores são métodos especiais que servem para inicializar objetos no momento da sua criação. Eles *possuem o mesmo nome da classe* e *não possuem tipo de retorno*, nem mesmo `void`.

Características Essenciais:

- **Nome**: Sempre o mesmo nome da classe.
- **Retorno**: Sem tipo de retorno, nem `void`.
- **Modificadores de acesso**: Podem ser `public`, `private` ou `protected`.
- **Parâmetros**: Opcionalmente, podem receber parâmetros para inicializar os atributos do objeto.

Funcionalidades:

- **Inicialização de Atributos**: Permitem definir valores iniciais para os atributos de instância da classe.
- **Execução Automática**: São chamados automaticamente quando um novo objeto é criado com o operador `new`.
- **Sobrecarga**: É possível ter mais de um construtor com diferentes parâmetros, permitindo flexibilidade na criação de objetos.
- **Chamada ao Construtor da Superclasse**: O construtor da superclasse pode ser chamado explicitamente com a palavra-chave super.

Tipos de Construtores:

- **Construtor Padrão**: Não possui parâmetros e é usado para criar objetos com valores default.
- **Construtor com Parâmetros**: Permite inicializar os atributos do objeto com valores específicos no momento da criação.
- **Construtor de Cópia**: Cria um novo objeto a partir de outro já existente.

Exemplos:

```
class Carro {
  private String marca;
  private int ano;

  // Construtor padrão
  public Carro() {
    this.marca = "Fiat";
    this.ano = 2023;
  }

  // Construtor com parâmetros
  public Carro(String marca, int ano) {
    this.marca = marca;
    this.ano = ano;
  }

  // ...
}

// Criação de objetos usando construtores
Carro carro1 = new Carro(); // Usa o construtor padrão
Carro carro2 = new Carro("Volkswagen", 2024); // Usa o construtor com
parâmetros
```

Importância dos Construtores:

- Permitem controlar a inicialização dos objetos, garantindo que estejam em um estado consistente.
- Facilitam a criação de objetos com diferentes configurações.
- Promovem flexibilidade e reuso de código.

Equals e HashCode em Java: Uma Dupla Essencial para Coleções e Comparação de Objetos

Equals e **HashCode** são dois métodos fundamentais em Java que trabalham em conjunto para garantir a comparação eficiente e precisa de objetos em coleções e outras operações.

Equals:

- **Objetivo**: Verifica se dois objetos são iguais em termos de seus valores.
- **Implementação padrão**: Compara os endereços de memória dos objetos, o que é ineficiente e inconsistente.
- **Boa prática**: Sobrescrever o método equals para comparar os atributos relevantes do objeto.

Exemplo:

```
public class Pessoa {
    private String nome;
    private int idade;

    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (!(obj instanceof Pessoa)) {
            return false;
        }
        Pessoa other = (Pessoa) obj;
        return Objects.equals(nome, other.nome) && idade == other.idade;
    }
}
```

HashCode:

- **Objetivo**: Gera um código hash para um objeto, que pode ser usado para comparação rápida em *coleções*.
- **Implementação padrão**: Gera um código hash baseado no endereço de memória do objeto, que é ineficiente e inconsistente.
- **Boa prática**: Sobrescrever o método hashCode para gerar um código hash consistente com base nos atributos do objeto.

Exemplo:

```
public class Pessoa {
    private String nome;
    private int idade;

    @Override
    public int hashCode() {
        return Objects.hash(nome, idade);
    }
}
```

Observação: Aprofundaremos e HashCode no próximo semestre quando começarmos a estudar Vetores e Collections Framework.

Revisão: Encapsulamento (Getters e Setters)

O encapsulamento é um pilar fundamental da orientação a objetos em Java. Ele consiste em ocultar os detalhes internos de um objeto e expor apenas uma interface pública para sua manipulação. Essa prática oferece diversos benefícios, como:

Segurança: Protege os dados internos contra acessos e modificações indevidas, garantindo a integridade do objeto.

Manutenabilidade: Facilita a manutenção do código, pois os detalhes de implementação ficam ocultos, permitindo modificar o funcionamento interno sem afetar os usuários do objeto.

Reutilização: Permite criar objetos mais genéricos e reutilizáveis, pois a interface pública define um contrato claro de como o objeto pode ser usado.

Flexibilidade: Permite modificar a implementação interna do objeto sem afetar os clientes que o utilizam, desde que a interface pública permaneça a mesma.

Mecanismos de Encapsulamento em Java:

- **Modificadores de Acesso**: Permitem controlar o nível de acesso aos membros de uma classe (atributos e métodos).
    - `public` : Permite acesso de qualquer lugar.
    - `private` : Permite acesso apenas dentro da classe.
    - `protected` : Permite acesso dentro da classe e de classes filhas.
- **Atributos Privados**: Os atributos de um objeto geralmente são declarados como private para garantir que apenas os métodos da classe possam acessá-los diretamente.
- **Métodos Acessores (Getters e Setters)**: São métodos públicos que permitem ler e modificar os valores dos atributos privados.
    - `get` - Lê o valor de um atributo privado.
    - `set` - Modifica o valor de um atributo privado.

UML: Linguagem de Modelagem Unificada

A UML (Unified Modeling Language) é uma linguagem de modelagem visual para especificar, construir e documentar sistemas de software. Ela fornece um conjunto de elementos gráficos para representar os diferentes componentes de um sistema, bem como suas relações e comportamentos.

Principais características da UML:

- **Visualização**: A UML permite a criação de modelos visuais que facilitam a compreensão e a comunicação do design do sistema.
- **Precisão**: A UML fornece uma linguagem formal para definir os elementos do sistema, evitando ambiguidades e interpretações errôneas.
- **Extensibilidade**: A UML é uma linguagem aberta que pode ser extendida para atender às necessidades específicas de diferentes projetos.
- **Padronização**: A UML é um padrão internacionalmente reconhecido, o que facilita a troca de informações entre diferentes equipes de desenvolvimento.

Semântica da UML:

A UML define um conjunto de regras que determinam o significado dos elementos gráficos utilizados na modelagem. Essas regras garantem que os modelos sejam consistentes e precisos. A semântica da UML é baseada em três conceitos principais:

- **Classes**: Representam os tipos de objetos que existem no sistema.
- **Associações**: Representam as relações entre as classes.
- **Operações**: Representam as ações que podem ser realizadas pelos objetos.

Uso da UML com Java:

A UML pode ser utilizada para modelar sistemas Java em diferentes níveis de abstração. Os principais diagramas da UML para modelagem de sistemas Java são:

- **Diagramas de Classes**: Representam as classes do sistema, seus atributos, métodos e relações.
- **Diagramas de Sequência**: Representam a sequência de interações entre os objetos do sistema.
- **Diagramas de Caso de Uso**: Representam os diferentes casos de uso do sistema e como os usuários interagem com ele.

Na nossa disciplina o foco será o Diagrama de Classes.

Construindo um Diagrama de Classes

1. **Identifique as classes**:

Comece por identificar as principais classes do seu sistema. Uma classe representa um tipo de objeto que existe no sistema, como um cliente, um produto ou um pedido.

2. **Defina os atributos e métodos**:

Para cada classe, defina os atributos e métodos que ela possui. Os atributos representam as características dos objetos, enquanto os métodos representam as ações que os objetos podem realizar.

3. **Identifique as relações entre as classes**:

Identifique as relações entre as classes do seu sistema. As principais relações são:

- **Associação**: Uma relação entre duas classes que indica que os objetos de uma classe podem estar relacionados aos objetos da outra classe.
- **Agregação**: Uma relação entre duas classes em que uma classe (a classe agregada) é parte da outra classe (a classe composta).
- **Composição**: Uma relação forte entre duas classes em que a classe agregada não pode existir sem a classe composta.
- **Herança**: Uma relação entre duas classes em que uma classe (a classe filha) herda os atributos e métodos da outra classe (a classe pai).

4. **Utilize a notação UML**:

Utilize a notação UML para representar as classes, seus atributos, métodos e relações. A notação UML é um conjunto de símbolos e regras que garantem que os diagramas de classes sejam consistentes e fáceis de entender.

5. **Ferramentas para criar diagramas de classes**:

Existem diversas ferramentas disponíveis para criar diagramas de classes, como:

- **Ferramentas UML**: Existem diversas ferramentas UML disponíveis no mercado, como o Visual Paradigm, o Enterprise Architect e o Rational Rose.
- **IDEs (Integrated Development Environments)**: Alguns IDEs, como o Eclipse e o IntelliJ IDEA, oferecem suporte à criação de diagramas de classes.

6. **Dicas para criar diagramas de classes**:

- Comece com um diagrama simples e vá adicionando detalhes gradualmente.
- Use nomes claros e concisos para as classes, atributos e métodos.
- Evite diagramas muito complexos.
- Utilize a notação UML de forma consistente.
- Revise o diagrama com frequência para garantir que ele esteja correto e atualizado.