

Orientação à objetos

Classe

Uma classe é uma forma de definir um tipo de dado em uma linguagem orientada a objeto. Ela é formada por dados e comportamentos. Para definir os dados são utilizados os atributos, e para definir o comportamento são utilizados métodos.

Objeto (Instância)

Chama-se instância, de uma classe, um objeto cujo comportamento e estado são definidos pela classe. Instância é a concretização de uma classe. Em termos intuitivos, uma classe é como um "molde" que gera instâncias de um certo tipo; um objeto é algo que existe fisicamente e que foi "moldado" na classe.

Classe

FORM

Part A :

Part B :


Part C :

Part D :

☒
☒

SIGNATURE _____

[Signature]
APPROVED



Objeto (Instância)

FORM

Part A : Fábio Correa
Github.com/Fa2bio
Desenvolvedor Java

Part B : ABCDE
FGIH
JKLM


Part C :
01/04/23

Part D :

☒
☒

SIGNATURE _____

[Signature]
APPROVED



This e This()

Utilizamos a palavra `this` quando queremos nos referir a um atributo ou método de uma classe. O `this.` refere-se a um ATRIBUTO e o `this()` refere-se a um MÉTODO.

```
public class Data {  
    int dia; int mes; int ano;  
  
    Data( ){ this(01,04,2023); }  
  
    Data (int dia, int mes, int ano) { this.dia = dia; this.mes = mes; this.ano = ano; }  
}
```

Interfaces

Uma interface não é considerada uma Classe e sim uma Entidade. Não possui implementação, apenas assinatura, ou seja, apenas a definição dos seus métodos sem o corpo. Seus métodos são implicitamente Públicos e Abstratos. A partir de uma interface, não há como se criar uma instância e nem como criar um Construtor.

Utilidades

Quando criamos uma Interface, estamos basicamente criando um conjunto de métodos sem qualquer implementação ou corpo, que devem ser implementados ou herdados por outras classes. Uma Interface permite fracionar os comportamentos de diversos objetos, o que permite “dividir” uma classe em pequenos fragmentos que compõem um todo. É uma alternativa para uma implementação de herança múltipla em Java

Interface

Em Java, utilize o modificador interface para indicar uma interface.

```
public interface Carro {  
    void acelerar ( );  
}
```

Classes Abstratas

As classes abstratas devem conter pelo menos um método abstrato, que não tem corpo. É um tipo especial de classe que não há como criar instâncias dela. É usada apenas para ser herdada, funciona como uma superclasse. A grande vantagem é que força a hierarquia para todas as sub-classes.

Utilidades

Quando criamos uma classe Abstrata, estamos criando uma classe base que contém atributos e pode ter um ou mais métodos completos (com corpo), mas pelo menos um ou mais destes métodos devem ser incompletos (sem corpo, abstratos), isto caracteriza uma Classe Abstrata.

Classe abstrata x Classe concreta

Classes abstratas são utilizadas para organizar e simplificar uma hierarquia de generalização.

- Ideias “incompletas” que não podem se concretizar na forma de objetos

Classes concretas podem ser instanciadas

- Ideias “completas”, mesmo que parte dessas ideias sejam herdadas de superclasses abstratas

Classe Abstrata

Em Java, utilize o modificador abstract para indicar que uma classe ou método abstrato

```
public abstract class Veiculo {  
    public abstract int numRodas( );  
}
```

Classe Concreta

```
public class Carro extends Veiculo{  
    public int numRodas( ){  
        return 4;  
    }  
}
```

Classe final

Uma classe final não pode ser estendida.

- Mecanismo utilizado para impedir a implementação de desdobramentos não planejados em uma biblioteca

Em Java, utilize o modificador *final* para indicar que a classe é final.

```
public final class Integer extends Number {  
  
}
```

Encapsulamento

O encapsulamento é um dos principais pilares da orientação a objetos. Por meio dele, é possível proteger (ou não) informações sigilosas ou sensíveis

Motivação para Encapsulamento

Manter cada classe responsável por operações a elas atribuídas sem interferências externas, assim delimita-se as funções das instâncias de cada classe, além de possibilitar que cada classe faça bem aquilo de que está encarregada, tendo controle total sobre tais operações e realização de manutenção na classe sem que os usuários da classe em questão sejam afetados

Níveis de Encapsulamento

- Nível de classe determina a visibilidade a uma classe inteira
- Nível de membro determina a visibilidade de atributos e métodos de uma classe

Modificadores de Acesso

- Modificador não explícito – Aplicado a classes e a membros, define visibilidade total ao elemento dentro do pacote corrente
- public – Aplicado a classes e a membros e define visibilidade total ao elemento
- protected – Aplicado a membros, torna o elemento visível apenas a subclasses da classe corrente

Modificadores de Acesso

- `private` – Aplicado a membros, torna o elemento invisível a outras classes.

Polimorfismo

Classes distintas possuem métodos de mesmo nome de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto

Utilidade

Projetar e implementar sistemas que são facilmente extensíveis, facilitar a adição, na hierarquia de classes, de novas classes sem modificar as partes gerais do programa, alterações só são necessárias nas classes que exigem conhecimento direto das novas classes que adicionamos à hierarquia

Exemplo

- Imagine que a classe SerVivo tem o método respirar
- A classe Cachorro implementa um comportamento para respirar, enquanto que a classe Peixe implementa outro comportamento
- Cachorro e Peixe são SerVivo, então podemos invocar o método respirar do SerVivo e o polimorfismo atribuirá o comportamento correto a invocação

Polimorfismo Estático

Polimorfismo estático é a sobrecarga de um método. É a capacidade de um método possuir o mesmo nome, mas com assinaturas diferentes.

```
public class Area{  
    public int soma (int a, int b) {  
        return a + b;  
    }  
  
    public double soma (double a, double b){  
        return a + b;  
    }  
}
```

Polimorfismo Dinâmico

A partir da herança é possível ter o polimorfismo dinâmico. Exemplo: Digamos temos uma classe Carro e uma classe Civic (que é um carro). Podemos instanciar um Civic a partir de

```
public class TesteCarro {  
    public static void main (String [] args){  
        Carro c1 = new Carro (0,10);  
        Carro c2 = new Civic(0,10,"Dual");  
    }  
}
```

Como Civic é um tipo de carro, está definido o polimorfismo dinâmico.

Relação one-to-one

No relacionamento OneToOne, **um item pode pertencer a apenas um outro item, é uma ligação um para um.**

```
public class Pessoa {  
    private Endereco endereco;  
}
```

Relação One-to-Many

No relacionamento One-to-Many, **um item pode pertencer a vários outros itens, é uma ligação um para muitos.**

```
public class Pessoa {  
    private List <Endereco> enderecos = new ArrayList<>();  
}
```

Relação Many-to-Many

No relacionamento Many-to-Many, **vários itens podem pertencer a vários outros itens, é uma ligação muitos para muitos.**

```
public class Aluno {  
    private String nome;  
    private List <Curso> cursos = new ArrayList<>();  
}  
  
public class Curso {  
    private String nome;  
    private List <Aluno> cursos = new ArrayList<>();  
}
```