VOLTAR

Abstração de dados

Em Java temos recursos super avançados que nos permitem criar ¿classes modelos¿ para outras, ou seja, uma classe que servirá de base para as outras em um determinado contexto. Estamos falando da abstração de dados em Java, ou seja, de classes abstratas e interfaces.

NESTE TÓPICO

- Classes Abstratas
- Interface
- Uma classe implementando mais de uma interface
- Resumo da aula
- Referências

Muitas vezes, ao invés de utilizarmos herança ou polimorfismo, precisamos apenas criar classes que servirão de modelo para outras criações. Estamos falando de classes abstratas e interface. Calma, ainda não vamos desenvolver interfaces gráficas, uma "interface" não tem relação com interface gráfica. E lembre-se que classes abstratas e interfaces são, ainda, tipos de polimorfismo.

Classes Abstratas

A criação de classes abstratas é um recurso bastante poderoso da linguagem. Não se preocupe, o conceito de objeto abstrato em Java é simples!

Imagine que você foi contratado(a) para desenvolver um sistema acadêmico, onde há alunos (o principal!), professores, funcionários administrativos (como coordenadores, inspetores de corredor, programadores para as ferramentas online etc.), entre outros. O que todos estes papéis no sistema têm em comum? Podem referir-se a uma única entidade, chamada "Pessoa", ou seja, todas as classes podem ser herdades de uma única classe "Pessoa" que já possui, em sua implementação, a base para as características e comportamentos (atributos e métodos) que são comuns de cada um dos

participantes do modelo. Em outras palavras, cada uma das classes do software, que se refere a um papel no sistema acadêmico, pode ser herdada de "Pessoa", certo?

Muito bem. Agora a pergunta é: Faz sentido ter (ser possível), no sistema, a instância de uma classe "Pessoa"? Não, pois todas as instâncias serão das classes herdadas e não da classe genérica "Pessoa". Em outras palavras, neste cenário do sistema acadêmico, é inadmissível existir a instância de uma classe "Pessoa", apenas podem existir instâncias de classes como "Aluno", "Professor" etc. É ai que usaremos as classes abstratas.

Classes abstratas, em Java, são classes modelo, ou seja, elas podem ser herdadas, mas todas as funcionalidades devem ser, obrigatoriamente, implementadas pela classe que está herdando. Na programação, é impossível criarmos uma instância de uma classe abstrata.

Resumindo: Se você tem uma classe abstrata, todas as classes que herdarem dela terão de reimplementar todos os métodos, pois ela funciona como modelo.

Para implementarmos classes abstratas, começaremos a utilizar uma nova palavra reservada: **abstract**.

Vamos codificar o cenário acadêmico, onde temos uma "PessoaGenerica" abstrata e uma classe aluno, herdando dessa casse seus atributos e métodos modelo. Uma possível implementação deste cenário pode ser vista abaixo:

Classe PessoaGenérica:

```
abstract class PessoaGenerica {
    String nome, cpf, rg, dtNascimento;

    public abstract String getDados();
}
```

Classe Aluno:

```
class Aluno extends PessoaGenerica {

    //Construtor
    public Aluno() {
    }

    @Override
    public String getDados() {
        String dados = "";
        dados += "\nNome: " + nome;
        dados += "\nCPF: " + cpf;
        dados += "\nRG: " + rg;
        dados += "\ndtNascimento: " + dtNascimento;
        return dados;
    }
}
```

Crie, por conta própria, a classe principal, que cria a instância de um Aluno, seta seus dados e imprime na tela os dados através da chamada ao método getDados().

Repare que a classe "PessoaGenerica" é abstrata e seu método de retornar dados também. Isso quer dizer que, se ele precisará ser reimplementado pela classe que herda, ele não precisa de um corpo, ou seja, basta a declaração do método. Lembre-se que, mesmo a classe "PessoaGenerica" sendo abstrata, seus atributos continuam sendo passados para as classes filhas (herdeiras).

É importante você perceber que nem todos os métodos precisam ser abstratos. Se você implementar um método concreto (sem a palavra reservada abstract na frente dele), esse método não precisará ser reimplementado pelas classes filhas.

Interface

Como mencionado anteriormente, não confunda classes tipo interface com interface gráfica, que é assunto para outra aula, só dela.

Diferentemente das classes abstratas, as interfaces podem conter apenas métodos abstratos (sem corpo, apenas a assinatura), ou seja, ela deve obrigatoriamente, forçar a reimplementação de seus métodos pelas classes filhas. Segundo Teruel, essa é uma vantagem da orientação a objetos, pois torna o código ainda mais reutilizável.

Para declararmos uma interface, não deve-se usar a palavra reservada "class", ou seja, a palavra reservada "interface" (que é igual em inglês e português, ufa!), vai no lugar da palavra "class". Veja abaixo um exemplo de uma interface de operações matemáticas (Teruel):

```
1.  public interface Operacoes{
2.
3.       //Essa é a declaração de um método sem corpo e abstrato
4.      abstract double calcularMedia(double n1, double n2);
5.  }
```

Assim como classes abstratas, as interfaces não podem ser instânciadas, ou seja, apenas herdadas por outras classes. Contudo, para classes implementarem interfaces, não se utiliza a palavra extends, mas sim **implements**. Veja um exemplo, abaixo, da classe "Professor" implementando a interface "Operacoes", para entender melhor:

```
1.  public class Professor implements Operacoes{
2.
3.      @Override
4.      public double calcularMedia(double n1, double n2){
5.              return (n1 + n2) / 2;
6.       }
7.  }
```

Note que houve obrigatoriedade na classe "Professor" para a implementação do método de cálculo da média. O compilador do Java não permitiria que a classe professor fosse compilada antes de todos os métodos serem sobrescritos (reimplementados).

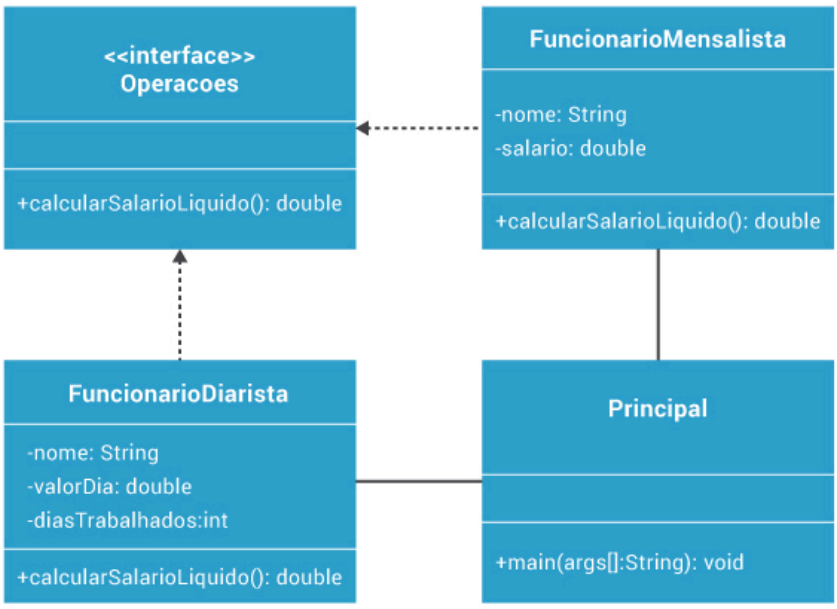
Você deve estar se perguntando: Mas quando que usarei interfaces? Bem, segundo Teruel:

- Quando deseja obrigar que um conjunto de classes de diferentes origens implemente um ou mais métodos de maneiras diferentes;
- Quando uma subclasse que já está herdando métodos e atributos de uma superclasse em relações de herança precisa herdar (e implementar) métodos de outro lugar. Como Java não permite herança múltipla nas versões anteriores á 8 (na versão 8 já permite), o uso de interfaces, neste caso, pode ser uma boa alternativa.

Para facilitar o entendimento, vamos programar!

Imagine, agora, que você foi contratado(a) para desenvolver uma aplicação de folha de pagamento de funcionários da empresa, contudo, esta possui dois tipos de funcionários: Diaristas e Mensalistas. O cálculo da folha de pagamento de cada um deles é diferente, pois o diarista recebe pelos dias trabalhados e o mensalista pelo mês completo.

Vamos implementar, então, o seguinte cenário (Teruel):



Modelo visual do exemplo de interface a ser implementado

Fonte: Teruel, E. C., 2015, Programação Orientada a Objetos com Java - sem mistérios - 1ª Ed., Editora Uninove

Isso quer dizer que você terá três classes (Principal, FuncionarioDiarista e FuncionarioMensalista) e uma interface (Operacoes). Lembre-se, novamente, que para cada um destes (classes ou interfaces) é preciso criar

um novo arquivo no projeto.

Vamos começar pela interface "Operacoes". Veja o código desta interface:

```
public interface Operacoes {

    //assinatura do método abstrato (obrigatório) para cálculo do salário liquido
    public double calcularSalarioLiquido();
}
```

Pronto. Temos a interface implementada. Agora vamos implementar as classes de negócio, que são a "FuncionarioDiarista" e "FuncionarioMensalista". Uma possível codificação de cada uma delas pode ser:

```
public class FuncionarioMensalista implements Operacoes {

    //Atributos:
    String nome;
    double salario;

    //Construtor que recebe parâmetros de entrada e seta localmente
    public FuncionarioMensalista(String nome, double salario) {
        this.nome = nome;
        this.salario = salario;
    }

    //Sobrescrevendo (obrigatoriamente) o método de cálculo do salário:
    @Override
    public double calcularSalarioLiquido() {
        return this.salario - this.salario * 27.5 / 100;
    }
}
```

E...:

```
public class FuncionarioDiarista implements Operacoes {

    //Atributos:
    String nome;
    double valorDaHora;
    int diasTrabalhados;

    //Construtor
    public FuncionarioDiarista(String nome, double valorDaHora, int diasTrabalhados)
{
        this.nome = nome;
        this.valorDaHora = valorDaHora;
        this.diasTrabalhados = diasTrabalhados;
    }

    //Sobrescrevendo (obrigatoriamente) o método de cálculo do salário:
    @Override
    public double calcularSalarioLiquido() {
        return (8 * valorDaHora) * diasTrabalhados;
    }
}
```

Vamos, agora, implementar a classe Principal, que faz tudo isso funcionar, lendo os dados de um funcionário mensalista, depois de um funcionário diarista e mostrando o salário liquido de cada um deles no console.

```
1.  import java.util.Scanner;
2.
3.  public class Principal {
4.
5.      public static void main(String args[]) {
6.          Scanner teclado = new Scanner(System.in);
7.          //Variáveis auxiliares:
8.          String nomeAux;
9.          double salarioAux;
10.         int diasAux;
11.
12.         //Le os dados para um funcionário mensalista:
13.         System.out.print("Qual o nome do funcionário mensalista? ");
14.         nomeAux = teclado.next();
15.
16.           System.out.print("Qual o valor do salário bruto do funcionário mensalist
    a?");
17.         salarioAux = teclado.nextDouble();
18.
19.         //Constrói um funcionário mensalista na memória (instância)
20.         FuncionarioMensalista fM = new FuncionarioMensalista(nomeAux, salarioAux);
21.
22.         //Le os dados para um funcionário diarista:
23.         System.out.print("\nQual o nome do funcionário diarista? ");
24.          nomeAux = teclado.next(); //Pode usar a mesma variável pois o funcionário me
    nsalista já foi construído.
25.
26.         System.out.print("Qual o valor da hora deste funcionário diarista?");
27.         salarioAux = teclado.nextDouble();
28.
29.         System.out.print("Quantos dias ele trabalhou?");
30.         diasAux = teclado.nextInt();
31.
32.         //Cria a instância de um funcionário diarista na memória, já com os dados:
33.          FuncionarioDiarista fD = new FuncionarioDiarista(nomeAux, salarioAux, diasAu
    x);
34.
35.         //Agora vamos imprimir os dados de cada um e o salário bruto (calculado):
36.         System.out.println("Dados e salário do Funcionário mensalista: ");
37.         System.out.println("\tNome: " + fM.nome);
38.         System.out.println("\tSalário Bruto: " + fM.calcularSalarioLiquido());
39.
40.         System.out.println("------\nDados e salário do Funcionário diarista: ");
41.         System.out.println("\tNome: " + fD.nome);
42.         System.out.println("\tSalário Bruto: " + fD.calcularSalarioLiquido());
43.     }
44. }
```

E o resultado da execução deste exemplo, pode ser visto abaixo:

```
aída - ExemploInterface (run)
run:
Qual o nome do funcionário mensalista? Josefino
Qual o valor do salário bruto do funcionário mensalista?12000

Qual o nome do funcionário diarista? Josefina
Qual o valor da hora deste funcionário diarista?150
Quantos dias ele trabalhou?28
Dados e salário do Funcionário mensalista:
        Nome: Josefino
        Salário Bruto: 8700.0
------
Dados e salário do Funcionário diarista:
        Nome: Josefina
        Salário Bruto: 33600.0
CONSTRUÍDO COM SUCESSO (tempo total: 31 segundos)
```

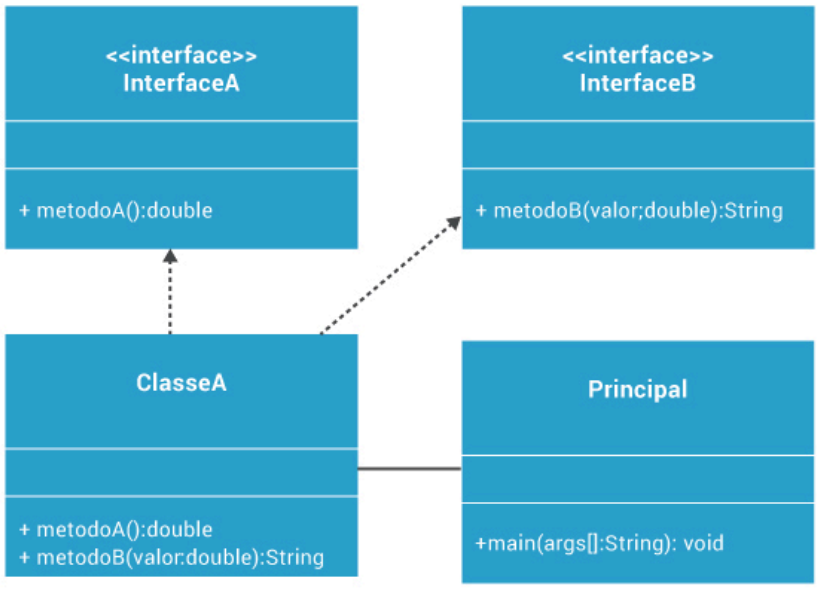
Resultado da execução do exemplo de interface

Calma! Está quase acabando! Antes de finalizarmos, vamos entender melhor o conceito quando uma classe implementa mais de uma interface, que é essencial e extremamente valioso na hora de programar.

Uma classe implementando mais de uma interface

Conforme mencionado anteriormente, uma classe pode implementar mais uma interface, simultaneamente. Este é um ótimo recurso da orientação a objetos pois permite criar quantas interfaces forem necessárias e usa-las conforme a necessidade.

Para exemplificar, vamos adota o seguinte modelo para implementação (Teruel):



Modelo visual do exemplo de multiplas interfaces a ser implementado

Fonte: Teruel, E. C., 2015, Programação Orientada a Objetos com Java - sem mistérios - 1ª Ed., Editora Uninove

Resumidamente: Temos duas interfaces (A e B), com métodos distintos e uma classe (A) que implementará essas interfaces e seus métodos. Temos a classe principal para testarmos e vermos isso tudo funcionando.

Sempre que uma classe irá implementar mais de uma interface, é preciso colocar na declaração da classe a palavra reservada **implementa** e separar o nome de cada interface por vírgula, assim:

```
1.  public class ClasseA implements interfaceA, interfaceB
```

> ## É IMPORTANTE LEMBRAR
>
> Lembre-se que sempre que uma classe implementar mais de uma interface, TODOS os métodos das duas interfaces deverão ser sobrescritos.

Teremos, então, a seguinte codificação:

Interface A:

```
1.  public interface InterfaceA {
2.
3.      public double metodoA();
4.  }
```

Interface B:

```
1.  public interface InterfaceB {
2.
3.      public String metodoB(double valor);
4.  }
```

Classe A:

```
public class ClasseA implements InterfaceA, InterfaceB {

    //Reimplementando o métodoA, da interface A
    @Override
    public double metodoA() {
        return 10.5;
    }

    //Reimplementando o métodoB, da interface B
    @Override
    public String metodoB(double valor) {
        if (valor <= 10) {
            return "Valor válido";
        } else {
            return "Valor inválido";
        }
    }
}
```

E, finalmente, a classe Principal, apenas para testar:

```
public class Principal {
    public static void main(String args[]) {
        ClasseA c = new ClasseA();
        double v = c.metodoA();
        String x = c.metodoB(10);
        System.out.println(x);
        System.out.println(v);
    }
}
```

Execute o projeto e veja o resultado da execução. Note que a reimplementação de cada um dos métodos foi chamada pela classe "Principal", reforçando o conceito de que, sempre que usamos mais de uma interface, é preciso sobrescrever todos os seus métodos.

Resumo da aula

Nesta aula você aprendeu conceitos muito importantes de Java que só são possíveis por conta da orientação a objetos. Os principais conceitos absorvidos aqui são:

- Classes abstratas servem de modelo para classes que a herdam. As classes abstratas podem possuir (ou não) métodos abstratos. Os métodos abstratos precisam, obrigatoriamente, serem sobrescritos nas classes filhas.
- Interfaces são modelos de classes 100% abstratas, ou seja, todos os métodos devem ser abstratos e, portanto, sem o corpo (apenas a assinatura). Interfaces são extremamente úteis quando precisamos de uma classe modelo e não podemos esquecer de implementar nenhum de seus métodos.
- Classes podem implementar mais de uma interface, simultaneamente, auxiliando, e muito, no reaproveitamento de código.

Chegamos ao final de mais uma aula. Não deixe de implementar os exemplos aqui descritos e estudar bastante estes importantes conceitos de Java. A primeira certificação da linguagem possui ao menos uma questão sobre cada um dos conceitos apresentados. Bons estudos e boa programação.

Quiz

Exercício Final

Abstração de dados

INICIAR

Referências

Deitei P. e Deitel H., 2010, Java : Como programar, 8ª Edição, Pearson Pretice Hall

Teruel, E. C., 2015, Programação Orientada a Objetos com Java - sem mistérios - 1ª Ed., Editora Uninove

Schildt, H., 2015, Schildt, Java para iniciantes : crie, compile e execute programas Java rapidamente, Bookman

Avalie este tópico

ANTERIOR
Herança e Polimorfismo

Biblioteca (https://www.uninove.br/conheca-a-

Índice

PRÓXIMO
Encapsulamento

Ajuda? (https://ava.un idCurso=)

uninove/biblioteca/sobre-a-biblioteca/apresentacao/)
Portal Uninove (http://www.uninove.br)
Mapa do Site