Programaçãolll:
Classes abstratas e Herança Múltipla

Profª. Tainá Isabela

Classes abstratas

Algumas classes são usadas apenas para agrupar características comuns a diversas classes e, então, ser herdada por outras classes. Tais classes são conhecidas como classes **abstratas**.

As classes que não são abstratas são conhecidas como classes **concretas**. As classes concretas podem ter instâncias diretas, ao contrário das classes abstratas que só podem ter **instâncias indiretas**, ou seja, apesar de a classe abstrata não poder ser instanciada, ela deve ter subclasses concretas que por sua vez podem ser instanciadas.

Classes abstratas

Às vezes é útil declarar classes – chamadas classes abstratas – para as quais você nunca pretende criar objetos. Como elas só são usadas como superclasses em hierarquias de herança, são chamadas superclasses abstratas. Essas classes não podem ser usadas para instanciar objetos, porque são incompletas.

```java
package banco;

public abstract class Conta {

    private int numero;
    private String nome_titular;
    protected double saldo;

    public Conta(int numero, String nome_titular, double saldo) {
        this.numero = numero;
        this.nome_titular = nome_titular;
        this.saldo = saldo;
    }

    public Conta(int numero, String nome_titular) {
        this.numero = numero;
        this.nome_titular = nome_titular;
        saldo = 0;
    }
```

```java
package banco;

public class UsaClasseAbstrata {

    public static void main(String[] args) {
        Conta c1 = new Conta(1,"Ze");//ERRO!
        ContaEspecial c2 = new ContaEspecial(2,"João",100);
        c1 = new ContaPoupanca(1,"Ze");
    }
```

Métodos abstratos

Em algumas situações as classes abstratas podem ser utilizadas para prover a definição de métodos que devem ser implementados em todas as suas subclasses, sem apresentar uma implementação para esses métodos. Tais métodos são chamados de **métodos abstratos**.

Toda classe que possui pelo menos um método abstrato é uma classe abstrata, mas uma classe pode ser abstrata sem possuir nenhum método abstrato.

public abstract boolean sacar(double valor);

```java
public class ContaEspecial extends Conta
{
    private double limite;

    @Override
    public boolean sacar(double valor){
        if (valor <= this.limite + this.saldo) {
            this.saldo -= valor;
            return true;
        }
        else{
            return false;
        }
    }
```

```java
public class ContaPoupanca extends Conta
{
    @Override
    public boolean sacar(double valor){
        if(this.getSaldo() >= valor){
            this.saldo -= valor;
            return true;
        }else {
            return false;
        }
    }
```

Para que uma subclasse de uma classe abstrata seja concreta, ela deve obrigatoriamente apresentar implementações concretas para todos os métodos abstratos de sua superclasse.

Por exemplo, se o método sacar não fosse implementado na classe ContaEspecial, essa classe teria de ser abstrata ou ocorreria erro de compilação.

Herança múltipla

O conceito de herança múltipla torna possível que uma classe descenda de várias classes.

Java não implementa herança múltipla por opção. Isso ocorre porque a herança múltipla pode nos gerar situações inusitadas.

Herança múltipla

- Teríamos uma superclasse Pessoa para agrupar as características comuns a todos as pessoas.
- Todo atleta é uma pessoa. Assim, Atleta herdaria de Pessoa (Em Java: Atleta **extends** Pessoa).
- Todo nadador é um atleta. Assim, Nadador herdaria de Atleta (Em Java: Nadador **extends** Atleta).
- Todo corredor é um atleta. Assim, Corredor herdaria de Atleta (Em Java: Corredor **extends** Atleta).
- Todo ciclista é um atleta. Assim, Ciclista herdaria de Atleta (Em Java: Ciclista **extends** Atleta).
- Todo triatleta é nadador, corredor e ciclista. Assim Triatleta deveria herdar de Nadador, Corredor e Atleta.

Interfaces

Uma interface pode ser vista como um **conjunto de declarações** de métodos, sem as respectivas implementações.

Uma interface é parecida com uma classe; porém, em uma interface, todos os métodos são públicos e abstratos e todos os atributos são públicos, estáticos e constantes.

A sintaxe para criar uma interface é muito parecida com a sintaxe para criar uma classe: **public interface <nome_da_interface>.**

Interfaces

O problema de herança múltipla apresentado na seção anterior pode ser resolvido com a criação de:

- Quatro interfaces: Atleta, Corredor, Nadador e Ciclista;
- Duas classes: Pessoa e Triatleta.

```java
package interfaces;
public interface Atleta {
    public static final int i = 0;
    public abstract void aquecer();
}
```

```java
package interfaces;
public interface Nadador extends Atleta {
    public void nadar();
}
```

```java
package interfaces;
public interface Corredor extends Atleta {
    public void correr();
}
```

```java
package interfaces;
public interface Ciclista extends Atleta {
    public void correrDeBicicleta();
}
```

```java
package interfaces;
public class Pessoa {
    private String nome, endereco;
    public Pessoa(String nome) {
        this.setNome(nome);
    }
    public String getEndereco() {
        return endereco;
    }
    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

```java
package interfaces;
public class Triatleta extends Pessoa implements Nadador, Corredor, Ciclista {
    public Triatleta(String nome) {
        super(nome);
    }
    public void aquecer() {
        System.out.println(this.getNome() + " está aquecendo");
    }
    public void nadar() {
        System.out.println(this.getNome() + " está nadando");
    }
    public void correr() {
        System.out.println(this.getNome() + " está correndo");
    }
    public void correrDeBicicleta() {
        System.out.println(this.getNome() + " está correndo de bike");
    }
}
```

O uso de interfaces é **recomendável** no desenvolvimento de sistemas para fornecer um contexto menos acoplado e mais simplificado de programação.

Vamos supor, por exemplo, que temos uma interface responsável pela comunicação com banco de dados; dessa forma, qualquer classe que implementar a interface responderá a todas as funcionalidades para acesso a banco.

Suponhamos que um novo banco seja elaborado e que desejemos fazer a troca do banco antigo por esse banco novo; será necessário apenas elaborar a classe que implemente a interface de acesso a esse banco novo e, ao invés de utilizarmos um objeto da classe antiga, utilizaremos um objeto da nova classe elaborada

Enumerações

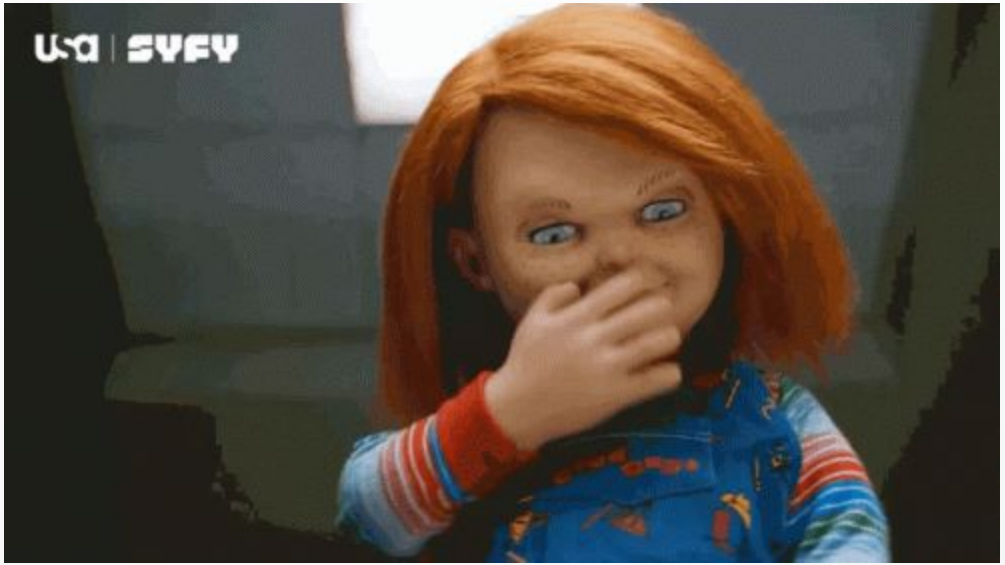
Os Enums são um tipo de dado especial que permitem que uma variável seja igual a um dos valores de um conjunto de constantes predefinidas.

É aconselhado o uso para sempre que um conjunto de constantes é necessário.

Ex: Status pré-definidos.

```
Public enum Tamanho { PEQUENO, MEDIO, GRANDE };
Tamanho medida = Tamanho.MEDIO;
```

Momento Questionário



Atividades

1. Crie uma classe abstrata chamada ContaBancaria com os seguintes métodos: public abstract boolean **sacar**(double valor); public abstract void **depositar**(double valor); Depois, crie duas subclasses concretas: **ContaCorrente** (com taxa de saque de R$ 1,00), **ContaPoupanca** (sem taxa de saque) Implemente os métodos abstratos e faça um programa principal para testar as operações de saque e depósito.

2. Crie um enum chamado **NivelAcesso** com os valores BASICO, INTERMEDIARIO e ADMIN. Em seguida, crie uma classe Usuario que possua: **nome**, **nível de acesso** (do tipo NivelAcesso). No programa principal, crie **três usuários** com diferentes níveis de acesso e exiba mensagens personalizadas com base no nível.

3. Crie interfaces para **Corredor**, **Nadador** e **Ciclista**, cada uma contendo um método representativo (correr(), nadar(), pedalar()). Depois, crie uma classe **Triatleta** que implemente as três interfaces e exiba mensagens apropriadas para cada método. Crie um programa principal para instanciar Triatleta e chamar os **três métodos**.