

Questões – Interfaces e Classes Abstratas em Java

Questões objetivas

1. Sobre uma classe abstrata em Java, assinale a alternativa correta:

- A) Pode ser instanciada normalmente utilizando `new`.
- B) Deve obrigatoriamente possuir apenas métodos abstratos.
- C) Pode possuir atributos, métodos concretos e métodos abstratos.
- D) Não pode possuir construtores.
- E) Pode implementar apenas uma interface.

2. Considere o código:

```
abstract class Animal {  
    public abstract void emitirSom();  
  
    public void dormir() {  
        System.out.println("Dormindo...");  
    }  
}
```

Sobre a classe `Animal`, é correto afirmar que:

- A) `Animal` pode ser instanciada diretamente.
- B) O método `emitirSom()` possui uma implementação obrigatória dentro de `Animal`.
- C) Uma classe concreta que herdar de `Animal` deverá implementar `emitirSom()`.
- D) O método `dormir()` também deve ser implementado pelas subclasses.
- E) Classes abstratas não podem possuir métodos concretos.

3. Analise:

```
interface Pagavel {  
    void pagar();  
}
```

Qual classe implementa corretamente essa interface?

A)

```
class Conta implements Pagavel {  
}
```

B)

```
class Conta extends Pagavel {  
    public void pagar() {}  
}
```

C)

```
class Conta implements Pagavel {  
    public void pagar() {}  
}
```

D)

```
class Conta extends Pagavel {  
}
```

E)

```
interface Conta implements Pagavel {  
}
```

4. Qual palavra-chave é utilizada para indicar que uma classe implementa uma interface?

- A) extends
- B) implements
- C) interface
- D) abstract
- E) inherit

5. Considere:

```
abstract class Forma {  
    public abstract double calcularArea();  
}  
  
class Retangulo extends Forma {  
  
    private double largura;  
    private double altura;  
  
    public Retangulo(double largura, double altura) {  
        this.largura = largura;  
        this.altura = altura;  
    }  
  
    @Override  
    public double calcularArea() {  
        return largura * altura;  
    }  
}
```

Qual é o objetivo principal do método `calcularArea()` declarado como abstrato?

- A) Impedir que `Retangulo` tenha atributos.
- B) Definir um comportamento que deverá ser implementado pelas subclasses.
- C) Permitir a criação de objetos da classe `Forma`.
- D) Transformar `Forma` em uma interface.
- E) Impedir o uso de polimorfismo.

6. Sobre interfaces em Java, assinale a alternativa correta:

- A) Uma classe pode implementar apenas uma interface.
- B) Interfaces não podem possuir métodos.
- C) Uma classe pode implementar múltiplas interfaces.
- D) Interfaces sempre devem possuir atributos privados.
- E) Uma interface deve obrigatoriamente possuir um construtor.

7. Analise o código:

```
interface Autenticavel {
    boolean autenticar();
}

interface Logavel {
    void registrarLog();
}

class Usuario implements Autenticavel, Logavel {

    @Override
    public boolean autenticar() {
        return true;
    }

    @Override
    public void registrarLog() {
        System.out.println("Log registrado");
    }
}
```

O código demonstra principalmente:

- A) Herança múltipla de classes.
- B) Implementação de múltiplas interfaces.
- C) Herança de uma classe abstrata.
- D) Encapsulamento de atributos.
- E) Sobrecarga de métodos.

8. Qual das alternativas apresenta uma declaração válida de classe abstrata?

A)

```
abstract class Veiculo {
}
```

B)

```
class abstract Veiculo {
}
```

C)

```
abstract Veiculo class {
}
```

D)

```
class Veiculo abstract {
}
```

E)

```
interface abstract Veiculo {
}
```

Questões de análise de código

9. Observe o código:

```
abstract class Funcionario {  
    protected String nome;  
  
    public Funcionario(String nome) {  
        this.nome = nome;  
    }  
  
    public abstract double calcularSalario();  
}  
  
class Gerente extends Funcionario {  
    private double salario;  
  
    public Gerente(String nome, double salario) {  
        super(nome);  
        this.salario = salario;  
    }  
  
    @Override  
    public double calcularSalario() {  
        return salario;  
    }  
}
```

Responda:

- a) Por que a classe `Funcionario` foi declarada como `abstract`?
- b) Por que o método `calcularSalario()` foi declarado como abstrato?
- c) É possível executar:

```
Funcionario f = new Funcionario("João");
```

Justifique.

- d) O código abaixo é válido?

```
Funcionario f = new Gerente("Maria", 5000);
```

Explique qual conceito de POO está sendo utilizado.

10. Analise:

```
interface Veiculo {  
    void acelerar();  
    void frear();  
}  
  
class Carro implements Veiculo {  
    @Override  
    public void acelerar() {
```

```

        System.out.println("Carro acelerando");
    }

    @Override
    public void frear() {
        System.out.println("Carro freando");
    }
}

```

Responda:

- a) Qual é a função da interface `Veiculo`?
- b) Por que a classe `Carro` utiliza `implements`?
- c) O que aconteceria se `Carro` não implementasse o método `frear()`?
- d) É possível criar:

```
Veiculo v = new Carro();
```

Justifique.

Questões discursivas

11. Explique, com suas próprias palavras, a diferença entre uma **classe abstrata** e uma **interface** em Java.

Apresente pelo menos **duas diferenças** entre elas.

12. Explique a seguinte afirmação:

"Uma classe abstrata representa uma base comum para suas subclasses, enquanto uma interface define um contrato que uma classe deve cumprir."

Dê um exemplo em Java para justificar sua resposta.

13. Considere o seguinte cenário:

Uma empresa possui diferentes tipos de funcionários:

- `Funcionario`
- `Programador`
- `Gerente`
- `Vendedor`

Todos possuem um nome, mas o cálculo do salário pode ser diferente para cada tipo.

Crie uma solução utilizando uma **classe abstrata** `Funcionario`, contendo:

- atributo `nome`;
- construtor;
- método abstrato `calcularSalario()`;
- pelo menos duas subclasses que implementem o cálculo do salário de maneiras diferentes.

14. Considere o seguinte cenário:

Um sistema possui diferentes formas de pagamento:

- cartão;
- PIX;
- boleto.

Todas as formas de pagamento devem possuir uma operação chamada `pagar()`.

Crie uma **interface Pagamento** e pelo menos três classes que a implementem.

Demonstre também como utilizar polimorfismo para executar o método `pagar()`.

15. – Questão de comparação

Considere as duas estruturas:

```
abstract class Animal {
    public abstract void emitirSom();

    public void dormir() {
        System.out.println("Dormindo...");
    }
}
```

e

```
interface Animal {
    void emitirSom();
}
```

Explique as principais diferenças entre essas duas abordagens.

Em sua resposta, considere:

- atributos;
- métodos concretos;
- métodos abstratos;
- herança;
- implementação de múltiplas interfaces;
- possibilidade de instanciação;
- utilização de polimorfismo.

Questão prática

16. – Desenvolvimento

Crie um programa Java para representar diferentes formas geométricas.

O programa deverá possuir:

Forma

- ├ Retangulo
- ├ Circulo
- └ Triangulo

A classe **Forma** deverá ser **abstrata** e possuir o método:

```
public abstract double calcularArea();
```

Cada classe filha deverá implementar seu próprio cálculo de área.

Depois, crie um `ArrayList<Forma>` contendo diferentes formas e percorra a lista chamando: `calcularArea()`

Pergunta: Qual conceito de Programação Orientada a Objetos está sendo demonstrado ao utilizar diferentes objetos através da referência **Forma**?

Questão-desafio

17. Um sistema de uma empresa possui funcionários que podem executar diferentes atividades.

Crie:

```
interface Autenticavel {  
    boolean autenticar();  
}
```

e:

```
abstract class Funcionario {  
    protected String nome;  
  
    public Funcionario(String nome) {  
        this.nome = nome;  
    }  
  
    public abstract double calcularSalario();  
}
```

Crie uma classe:

```
class Gerente extends Funcionario implements Autenticavel
```

que implemente os métodos necessários.

Depois responda:

- a) Qual é a relação entre **Gerente** e **Funcionario**?
- b) Qual é a relação entre **Gerente** e **Autenticavel**?
- c) Por que **Gerente** pode herdar atributos e métodos de **Funcionario**?
- d) Por que **Gerente** precisa implementar `autenticar()`?
- e) Quais conceitos de POO estão presentes nesse exemplo?

Gabarito – Interfaces e Classes Abstratas em Java

Questões objetivas

1. C

Uma classe abstrata pode possuir atributos, métodos concretos, métodos abstratos e construtores.

2. C

Uma classe concreta que herda de `Animal` deve implementar o método abstrato `emitirSom()`.

3. C

```
class Conta implements Pagavel {  
    public void pagar() {  
    }  
}
```

4. B — `implements`

5. B

O método abstrato estabelece um comportamento que deverá ser implementado pelas subclasses.

6. C

Uma classe pode implementar várias interfaces.

7. B

O exemplo demonstra a implementação de múltiplas interfaces:

```
class Usuario implements Autenticavel, Logavel
```

8. A

```
abstract class Veiculo {  
}
```

Questões de análise de código

9.

a) Por que `Funcionario` foi declarada como `abstract`?

Porque `Funcionario` representa uma classe-base que possui características comuns aos funcionários, mas não necessariamente representa um tipo concreto de funcionário que deva ser instanciado diretamente.

Além disso, ela possui o método abstrato:

```
public abstract double calcularSalario();
```

que deverá ser implementado pelas subclasses.

b) Por que `calcularSalario()` foi declarado como abstrato?

Porque o cálculo do salário pode ser diferente para cada tipo de funcionário.

A classe `Funcionario` define **o que deve ser feito**, mas deixa para as subclasses definir **como será feito**.

c) É possível executar?

```
Funcionario f = new Funcionario("João");
```

Não.

Classes abstratas não podem ser instanciadas diretamente.

d) O código abaixo é válido?

```
Funcionario f = new Gerente("Maria", 5000);
```

Sim.

Esse código demonstra **polimorfismo**.

A variável `f` possui o tipo da classe abstrata `Funcionario`, mas referencia um objeto concreto da classe `Gerente`.

10.

a) Qual é a função da interface `Veiculo`?

A interface define um **contrato** que determina que uma classe que a implemente deverá fornecer os comportamentos:

```
acelerar()  
frear()
```

b) Por que **Carro** utiliza **implements**?

Porque **Carro** está implementando o contrato definido pela interface **Veiculo**.

```
class Carro implements Veiculo
```

c) O que aconteceria se **Carro** não implementasse **frear()**?

Como **Carro** é uma classe concreta, ela deverá implementar todos os métodos abstratos da interface.

Caso não implemente **frear()**, ocorrerá erro de compilação.

Uma alternativa seria declarar **Carro** como **abstract**.

d) É possível fazer:

```
Veiculo v = new Carro();
```

Sim.

Esse é outro exemplo de **polimorfismo**.

A referência é do tipo **Veiculo**, enquanto o objeto é do tipo **Carro**.

Questões discursivas

11. Diferença entre classe abstrata e interface

Resposta esperada

Uma **classe abstrata** é utilizada quando queremos criar uma classe-base que compartilha características e comportamentos entre suas subclasses.

Ela pode possuir:

- atributos;
- construtores;
- métodos concretos;
- métodos abstratos.

Exemplo:

```
abstract class Animal {  
  
    protected String nome;  
  
    public Animal(String nome) {  
        this.nome = nome;  
    }  
  
    public void dormir() {  
        System.out.println("Dormindo");  
    }  
}
```

```
}    public abstract void emitirSom();  
}
```

Uma **interface** representa um contrato que as classes podem implementar.

Exemplo:

```
interface Autenticavel {  
    boolean autenticar();  
}
```

Uma classe pode implementar várias interfaces:

```
class Usuario implements Autenticavel, Logavel {  
}
```

Resposta resumida

Classe abstrata	Interface
Pode possuir atributos de instância	Pode declarar constantes
Pode possuir construtor	Não possui construtor
Pode possuir métodos concretos	Pode possuir métodos abstratos e, nas versões modernas do Java, <code>default</code> e <code>static</code>
Utiliza <code>extends</code>	Utiliza <code>implements</code>
Uma classe pode estender apenas uma classe	Uma classe pode implementar várias interfaces

12. Classe abstrata como base e interface como contrato

Resposta esperada

A classe abstrata pode fornecer características e comportamentos comuns:

```
abstract class Funcionario {  
    protected String nome;  
  
    public void registrarEntrada() {  
        System.out.println("Entrada registrada");  
    }  
  
    public abstract double calcularSalario();  
}
```

Nesse caso, todas as subclasses de `Funcionario` possuem `nome` e podem utilizar `registrarEntrada()`.

Já uma interface define um comportamento que uma classe deve fornecer:

```
interface Autenticavel {  
    boolean autenticar();  
}
```

Qualquer classe que implemente `Autenticavel` deverá fornecer `autenticar()`.

Portanto:

- **classe abstrata:** fornece uma base comum;
- **interface:** estabelece um contrato.

13. Funcionários

Uma solução possível:

```
abstract class Funcionario {  
    protected String nome;  
  
    public Funcionario(String nome) {  
        this.nome = nome;  
    }  
  
    public abstract double calcularSalario();  
}
```

Subclasse Programador:

```
class Programador extends Funcionario {  
    private double salario;  
  
    public Programador(String nome, double salario) {  
        super(nome);  
        this.salario = salario;  
    }  
  
    @Override  
    public double calcularSalario() {  
        return salario;  
    }  
}
```

Subclasse Gerente:

```
class Gerente extends Funcionario {  
    private double salario;  
    private double bonus;  
  
    public Gerente(String nome, double salario, double bonus) {  
        super(nome);  
        this.salario = salario;  
        this.bonus = bonus;  
    }  
  
    @Override  
    public double calcularSalario() {
```

```
        return salario + bonus;
    }
}
```

Conceitos demonstrados

- abstração;
- herança;
- sobrescrita de métodos;
- polimorfismo;
- classe abstrata.

14. Formas de pagamento

Interface:

```
interface Pagamento {
    void pagar();
}
```

PIX:

```
class PagamentoPix implements Pagamento {

    @Override
    public void pagar() {
        System.out.println("Pagamento realizado via PIX");
    }
}
```

Cartão:

```
class PagamentoCartao implements Pagamento {

    @Override
    public void pagar() {
        System.out.println("Pagamento realizado via cartão");
    }
}
```

Boleto:

```
class PagamentoBoleto implements Pagamento {

    @Override
    public void pagar() {
        System.out.println("Pagamento realizado via boleto");
    }
}
```

Uso polimórfico:

```
Pagamento pagamento;

pagamento = new PagamentoPix();
pagamento.pagar();
```

```
pagamento = new PagamentoCartao();
pagamento.pagar();
```

```
pagamento = new PagamentoBoleto();
pagamento.pagar();
```

Também seria possível utilizar uma coleção:

```
List<Pagamento> pagamentos = new ArrayList<>();

pagamentos.add(new PagamentoPix());
pagamentos.add(new PagamentoCartao());
pagamentos.add(new PagamentoBoleto());

for (Pagamento pagamento : pagamentos) {
    pagamento.pagar();
}
```

Conceitos esperados

- interface;
- implementação;
- polimorfismo;
- sobrescrita de métodos;
- abstração.

15. Comparação

Classe abstrata

```
abstract class Animal {

    public abstract void emitirSom();

    public void dormir() {
        System.out.println("Dormindo...");
    }
}
```

Pode possuir:

- atributos;
- construtores;
- métodos concretos;
- métodos abstratos.

Uma classe concreta pode herdar apenas de uma classe:

```
class Cachorro extends Animal {
}
```

Interface

```
interface Animal {
```

```

    void emitirSom();
}

```

Representa principalmente um contrato.

Uma classe pode implementar várias interfaces:

```

class Cachorro implements Animal, Domestico, Corredor {
}

```

Comparação esperada

Característica	Classe abstrata	Interface
Instanciação direta	Não	Não
<code>new</code> diretamente	Não	Não
Métodos abstratos	Sim	Sim
Métodos concretos	Sim	Sim, por exemplo <code>default</code>
Atributos de instância	Sim	Não da mesma forma que uma classe
Construtor	Sim	Não
Herança múltipla	Não para classes	Uma classe pode implementar várias interfaces
Palavra-chave	<code>extends</code>	<code>implements</code>
Polimorfismo	Sim	Sim

16. Formas geométricas

Uma possível solução:

```

abstract class Forma {

    public abstract double calcularArea();
}

```

Retângulo:

```

class Retangulo extends Forma {

    private double largura;
    private double altura;

    public Retangulo(double largura, double altura) {
        this.largura = largura;
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return largura * altura;
    }
}

```

Círculo:

```

class Circulo extends Forma {

```

```

    private double raio;

    public Circulo(double raio) {
        this.raio = raio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * raio * raio;
    }
}

```

Triângulo:

```

class Triangulo extends Forma {

    private double base;
    private double altura;

    public Triangulo(double base, double altura) {
        this.base = base;
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return (base * altura) / 2;
    }
}

```

Uso:

```

List<Forma> formas = new ArrayList<>();

formas.add(new Retangulo(10, 5));
formas.add(new Circulo(5));
formas.add(new Triangulo(10, 8));

for (Forma forma : formas) {
    System.out.println(forma.calcularArea());
}

```

Resposta da pergunta

O conceito demonstrado é principalmente o **polimorfismo**.

A variável:

Forma forma

pode referenciar objetos de diferentes subclasses:

Retangulo
Circulo
Triangulo

Quando:

forma.calcularArea();

é executado, Java chama a implementação correspondente ao tipo real do objeto.

17. Questão-desafio

Uma possível implementação:

```
interface Autenticavel {  
    boolean autenticar();  
}
```

Classe abstrata:

```
abstract class Funcionario {  
    protected String nome;  
  
    public Funcionario(String nome) {  
        this.nome = nome;  
    }  
  
    public abstract double calcularSalario();  
}
```

Classe Gerente:

```
class Gerente extends Funcionario implements Autenticavel {  
    private double salario;  
  
    public Gerente(String nome, double salario) {  
        super(nome);  
        this.salario = salario;  
    }  
  
    @Override  
    public double calcularSalario() {  
        return salario;  
    }  
  
    @Override  
    public boolean autenticar() {  
        System.out.println("Gerente autenticado");  
        return true;  
    }  
}
```

a) Qual é a relação entre Gerente e Funcionario?

Gerente **herda** de Funcionario.

```
class Gerente extends Funcionario
```

É uma relação de **herança**.

b) Qual é a relação entre Gerente e Autenticavel?

Gerente **implementa** a interface Autenticavel.

```
class Gerente extends Funcionario implements Autenticavel
```

c) Por que **Gerente** pode herdar atributos e métodos de **Funcionario**?

Porque **Gerente** é uma subclasse de **Funcionario**.

Por isso, ela pode utilizar os membros herdáveis da classe-pai, incluindo o atributo:

```
protected String nome;
```

d) Por que **Gerente** precisa implementar **autenticar()**?

Porque **Gerente** é uma classe concreta e implementa a interface:

```
Autenticavel
```

A interface estabelece que a classe deverá fornecer uma implementação para:

```
boolean autenticar();
```

e) Quais conceitos de POO estão presentes?

Espera-se que o aluno identifique:

- **Abstração** — **Funcionario** é uma classe abstrata;
- **Herança** — **Gerente** extends **Funcionario**;
- **Interface** — **Autenticavel**;
- **Implementação de interface** — **Gerente** implements **Autenticavel**;
- **Polimorfismo** — uma referência de **Funcionario** pode apontar para um **Gerente**;
- **Encapsulamento** — utilização de atributos e métodos com diferentes níveis de acesso;
- **Sobrescrita (@Override)** — implementação de **calcularSalario()** e **autenticar()**.