PROVA PARA SELEÇÃO DE BOLSISTA DE MONITORIA (10pts)

INF03068 - DESENVOLVIMENTO ORIENTADO A OBJETOS

Prof. Filipe Fernandes, PhD

Nome: Data: / / Nota:

ATENÇÃO! Nos códigos, desconsidere a necessidade do *import*.

1) Considere o seguinte código em Java: (2pts)

```
abstract class Veiculo {
    private String marca;


    public Veiculo(String marca) {
        this.marca = marca;
    }

    public String getMarca() {
        return marca;
    }

    public abstract int getMaxVelocidade();

    public void infoVeiculo() {
        System.out.println("Marca: " + getMarca());
        System.out.println("Velocidade Máxima: " + getMaxVelocidade() + " km/h");
    }
}

abstract class Carro extends Veiculo {
    public Carro(String marca) {
        super(marca);
    }

    public abstract void abrirPortaMalas();

    @Override
    public void infoVeiculo() {
        super.infoVeiculo();
        System.out.println("Tipo: Carro");
    }
}

class Sedan extends Carro {
    public Sedan(String marca) {
        super(marca);
    }

    @Override
    public int getMaxVelocidade() {
        return 200;
    }

    @Override
    public void abrirPortaMalas() {
        System.out.println("Porta-malas do Sedan aberto.");
    }
}

public class TesteAbstrato {
    public static void main(String[] args) {
        Veiculo veiculo = new Sedan("Toyota");
        veiculo.infoVeiculo();
        ((Carro) veiculo).abrirPortaMalas();
```

```
    }
}
```

Qual será a saída do programa ao executar o método *main*?

a)
```
Marca: Toyota
Velocidade Máxima: 200 km/h
Porta-malas do Sedan aberto.
```

b)
```
Marca: Toyota
Velocidade Máxima: 200 km/h
Tipo: Carro.
```

c)
```
Marca: Toyota
Velocidade Máxima: 200 km/h
Tipo: Carro
Porta-malas do Sedan aberto.
```

d) O programa não compila porque a classe *Carro* é abstrata e não pode ser instanciada.

e) O programa lança uma exceção *ClassCastException* ao tentar fazer o cast de veiculo para *Carro*.

2) Considere o código Java: (2pts)

```
class A {
    private String atributo1;
    private String atributo2;

    public A(String atributo1, String atributo2) {
        this.atributo1 = atributo1;
        this.atributo2 = atributo2;
    }

    public String getAtributo1() {
        return atributo1;
    }

    public String getAtributo2() {
        return atributo2;
    }
}

class B {
    private List<A> listaDeAs;

    public B() {
        listaDeAs = new ArrayList<>();
    }

    public void adicionarA(A a) {
        listaDeAs.add(a);
    }

    public List<A> buscarAsPorAtributo2(String valorAtributo2) {
        List<A> resultados = new ArrayList<>();
        for (A a : listaDeAs) {
            if (a.getAtributo2().equals(valorAtributo2)) {
                resultados.add(a);
            }
        }
        return resultados;
    }

    public void listarAs() {
        if (listaDeAs.isEmpty()) {
            System.out.println("A lista está vazia.");
```

```
        } else {
            for (A a : listaDeAs) {
                System.out.println("Atributo1: " + a.getAtributo1() + ", Atributo2: " + a.getAtributo2());
            }
        }
    }
}

public class C {
    public static void main(String[] args) {
        B b = new B();
        A a1 = new A("Valor1", "X");
        A a2 = new A("Valor2", "X");
        A a3 = new A("Valor3", "Y");

        b.adicionarA(a1);
        b.adicionarA(a2);
        b.adicionarA(a3);

        System.out.println("Objetos com Atributo2 = X:");
        List<A> resultados = b.buscarAsPorAtributo2("X");
        for (A a : resultados) {
            System.out.println(a.getAtributo1());
        }
    }
}
```

Qual será a saída do programa ao executar o método *main*?

a)
```
Objetos com Atributo2 = X:
Atributo1: Valor1, Atributo2: X
Atributo1: Valor2, Atributo2: X
Atributo1: Valor3, Atributo2: Y
```

b)
```
A lista está vazia.
Objetos com Atributo2 = X:
Valor1
Valor2
```

c)
```
Objetos com Atributo2 = X:
Valor1
Valor2
```

d) O programa não compila devido à ausência de um método *toString()* na classe A.

e) O programa lança uma *NullPointerException* ao tentar acessar a lista de objetos do tipo A em B.

3) Considere o código Java: (2pts)

```
class Animal {
    public void emitirSom() {
        System.out.println("Som de um animal");
    }
}

class Cachorro extends Animal {
    @Override
    public void emitirSom() {
        System.out.println("Latido");
    }

    public void correr() {
        System.out.println("Cachorro correndo");
    }
}

class Gato extends Animal {
```

```
    @Override
    public void emitirSom() {
        System.out.println("Miado");
    }
}

public class TestePolimorfismo {
    public static void main(String[] args) {
        Animal animal1 = new Cachorro();
        Animal animal2 = new Gato();

        animal1.emitirSom();
        animal2.emitirSom();

        // Código adicional
        if (animal1 instanceof Cachorro) {
            ((Cachorro) animal1).correr();
        }
    }
}
```

Qual das alternativas a seguir descreve corretamente o comportamento do programa quando o método main é executado?

a) A saída será:
```
Som de um animal
Som de um animal
Cachorro correndo
```
b) A saída será:
```
Latido
Miado
Cachorro correndo
```
c) O programa lança uma exceção *ClassCastException* ao tentar fazer o casting de *animal1* para *Cachorro*.
d) A chamada ao método *emitirSom()* em *animal2* resultará em uma saída "Som de um animal" porque o polimorfismo não se aplica a métodos sobrescritos em Java.
e) O programa não compila devido ao uso incorreto do operador *instanceof*.

4) Considere o seguinte código em Java: (2pts)

```
interface Corredor {
    void correr();
    default void aquecer() {
        System.out.println("Aquecimento do corredor.");
    }
}

interface Nadador {
    void nadar();
    default void aquecer() {
        System.out.println("Aquecimento do nadador.");
    }
}

class Triatleta implements Corredor, Nadador {
    @Override
    public void correr() {
        System.out.println("Triatleta correndo.");
    }

    @Override
    public void nadar() {
        System.out.println("Triatleta nadando.");
    }
```

```
    // Método de aquecimento específico implementado aqui
    @Override
    public void aquecer() {
        Corredor.super.aquecer();
        Nadador.super.aquecer();
        System.out.println("Aquecimento completo do triatleta.");
    }
}

public class TesteInterface {
    public static void main(String[] args) {
        Triatleta atleta = new Triatleta();
        atleta.correr();
        atleta.nadar();
        atleta.aquecer();
    }
}
```

Qual será a saída quando o método *main* for executado?

a)
```
Triatleta correndo.
Triatleta nadando.
Aquecimento do corredor.
Aquecimento completo do triatleta.
```

b)
```
Triatleta correndo.
Triatleta nadando.
Aquecimento do nadador.
Aquecimento completo do triatleta.
```

c)
```
Triatleta correndo.
Triatleta nadando.
Aquecimento do corredor.
Aquecimento do nadador.
Aquecimento completo do triatleta.
```

d) O código não compila devido ao conflito entre os métodos *aquecer()* das interfaces *Corredor* e *Nadador*.

e) O programa lança uma exceção *UnsupportedOperationException* durante a execução devido ao uso de múltiplos métodos default com o mesmo nome.

5) Considere que estamos implementando uma estrutura de diretórios e arquivos, onde tanto diretórios quanto arquivos devem ser tratados de forma unificada em uma hierarquia. Analise os trechos de código abaixo e escolha o que implementa corretamente. (2pts)

a)
```
interface Componente {
    void exibir();
}

class Arquivo implements Componente {
    private String nome;

    public Arquivo(String nome) {
        this.nome = nome;
    }

    @Override
    public void exibir() {
        System.out.println("Arquivo: " + nome);
    }
}

class Diretorio implements Componente {
    private String nome;
    private List<Arquivo> arquivos = new ArrayList<>();

    public Diretorio(String nome) {
```

```
        this.nome = nome;
    }

    public void adicionarArquivo(Arquivo arquivo) {
        arquivos.add(arquivo);
    }

    @Override
    public void exibir() {
        System.out.println("Diretório: " + nome);
        for (Arquivo arquivo : arquivos) {
            arquivo.exibir();
        }
    }
}
```

b)

```
abstract class Componente {

    public abstract void exibir();
}

class Arquivo extends Componente {
    private String nome;

    public Arquivo(String nome) {
        this.nome = nome;
    }

    @Override
    public void exibir() {
        System.out.println("Arquivo: " + nome);
    }
}

class Diretorio extends Componente {
    private String nome;
    private List<Arquivo> arquivos = new ArrayList<>();

    public Diretorio(String nome) {
        this.nome = nome;
    }

    public void adicionarArquivo(Arquivo arquivo) {
        arquivos.add(arquivo);
    }

    @Override
    public void exibir() {
        System.out.println("Diretório: " + nome);
        for (Arquivo arquivo : arquivos) {
            arquivo.exibir();
        }
    }
}
```

c)

```
interface Componente {

    void exibir();
}

class Arquivo implements Componente {
    private String nome;

    public Arquivo(String nome) {
        this.nome = nome;
    }
```

```
    @Override
    public void exibir() {
        System.out.println("Arquivo: " + nome);
    }
}

class Diretorio implements Componente {
    private String nome;
    private List<Diretorio> subDiretorios = new ArrayList<>();

    public Diretorio(String nome) {
        this.nome = nome;
    }

    public void adicionarDiretorio(Diretorio diretorio) {
        subDiretorios.add(diretorio);
    }

    @Override
    public void exibir() {
        System.out.println("Diretório: " + nome);
        for (Diretorio diretorio : subDiretorios) {
            diretorio.exibir();
        }
    }
}
```

d)

```
interface Componente {
    void exibir();
}

class Arquivo implements Componente {
    private String nome;

    public Arquivo(String nome) {
        this.nome = nome;
    }

    @Override
    public void exibir() {
        System.out.println("Arquivo: " + nome);
    }
}

class Diretorio implements Componente {
    private String nome;
    private List<Componente> componentes = new ArrayList<>();

    public Diretorio(String nome) {
        this.nome = nome;
    }

    public void adicionarComponente(Componente componente) {
        componentes.add(componente);
    }

    @Override
    public void exibir() {
        System.out.println("Diretório: " + nome);
        for (Componente componente : componentes) {
            componente.exibir();
        }
    }
}
```

e)

```
class Arquivo {
    private String nome;

    public Arquivo(String nome) {
        this.nome = nome;
    }

    public void exibir() {
        System.out.println("Arquivo: " + nome);
    }
}

class Diretorio {
    private String nome;
    private List<Arquivo> arquivos = new ArrayList<>();

    public Diretorio(String nome) {
        this.nome = nome;
    }

    public void adicionarArquivo(Arquivo arquivo) {
        arquivos.add(arquivo);
    }

    public void exibir() {
        System.out.println("Diretório: " + nome);
        for (Arquivo arquivo : arquivos) {
            arquivo.exibir();
        }
    }
}
```