Matéria: Programação

Assunto: Programação Orientada a Objetos em Java

Resumo Teórico do Assunto

Para resolver as questões de Programação Orientada a Objetos (POO) em Java, é fundamental compreender como as classes interagem, como os objetos são criados e como o comportamento é determinado em tempo de execução.

Programação Orientada a Objetos (POO) em Java: Conceitos Essenciais

A Programação Orientada a Objetos (POO) é um paradigma de programação baseado no conceito de "objetos", que podem conter dados (atributos) e código (métodos). Em Java, tudo é construído em torno de classes e objetos.

 # 1. Classes e Objetos

• **Classe:** É um **molde** ou **planta** para criar objetos. Ela define a estrutura (atributos) e o comportamento (métodos) que os objetos daquela classe terão.
* **Atributos (Variáveis de Instância):** São as características ou dados que um objeto possui. Ex: `int a; String nome;`.
* **Métodos:** São as ações ou comportamentos que um objeto pode realizar. Ex: `public int op1(int x);`.
• **Objeto (Instância):** É uma **ocorrência concreta** de uma classe. Quando você cria um objeto, você está instanciando a classe. Ex: `CAx o = new CBy();`.

 # 2. Pilares da POO (Relevantes para as questões)

• **Encapsulamento:** É o princípio de ocultar os detalhes internos de um objeto e expor apenas o que é necessário para interagir com ele. Isso é feito através de **modificadores de acesso**:
* **`public`**: Acessível de qualquer lugar.
* **`protected`**: Acessível dentro da própria classe, classes do mesmo pacote e subclasses (mesmo que em pacotes diferentes).
* **`private`**: Acessível apenas dentro da própria classe.
* **(default/package-private)**: Acessível apenas dentro do mesmo pacote.

• **Herança:** Permite que uma classe (subclasse ou classe filha) herde atributos e métodos de outra classe (superclasse ou classe pai). Isso promove a **reutilização de código**.
* Usa a palavra-chave `extends`. Ex: `public class CBy extends CAx`.

* A subclasse herda todos os membros `public` e `protected` da superclasse. Membros `private` não são herdados diretamente, mas podem ser acessados indiretamente via métodos `public` ou `protected` da superclasse.
* **`super` keyword:**
* `super(...)`: Usado para chamar um **construtor da superclasse**. Deve ser a primeira instrução no construtor da subclasse.
* `super.atributo`: Acessa um atributo da superclasse (útil em caso de sombreamento).
* `super.metodo()`: Chama um método da superclasse.

• **Polimorfismo:** Significa "muitas formas". Permite que objetos de diferentes classes sejam tratados como objetos de uma classe comum (sua superclasse).
* **Upcasting:** Atribuir uma instância de uma subclasse a uma variável de referência do tipo da superclasse. Ex: `CAx o = new CBy();`. O objeto `o` é do tipo `CBy`, mas a referência é do tipo `CAx`.
* **Sobrescrita de Métodos (Method Overriding):** Uma subclasse fornece uma implementação específica para um método que já é definido na sua superclasse.
* A assinatura do método (nome, tipo de retorno, número e tipo de parâmetros) deve ser a mesma.
* A resolução de qual método será chamado (o da superclasse ou o da subclasse) ocorre em **tempo de execução** (conhecido como **Dynamic Method Dispatch**), baseada no *tipo real do objeto*, não no tipo da referência.
* Aplica-se apenas a métodos de **instância** (não estáticos).
* **Ocultação de Métodos Estáticos (Static Method Hiding):** Quando uma subclasse define um método `static` com a mesma assinatura de um método `static` na superclasse.
* Não é sobrescrita. A resolução de qual método será chamado ocorre em **tempo de compilação**, baseada no *tipo da referência*, não no tipo real do objeto.
* **Sombreamento de Atributos (Field Hiding/Shadowing):** Quando uma subclasse declara um atributo com o mesmo nome de um atributo na superclasse.
* Ambos os atributos existem. O acesso ao atributo depende do *tipo da referência* que está sendo usada para acessá-lo. Se a referência é do tipo da superclasse, o atributo da superclasse é acessado; se é do tipo da subclasse, o atributo da subclasse é acessado.

 # 3. Construtores e Ordem de Inicialização

• **Construtor:** Um método especial usado para inicializar um objeto recém-criado. Tem o mesmo nome da classe e não possui tipo de retorno (nem mesmo `void`).
• **Bloco de Inicialização de Instância (`{ ... }`):** Um bloco de código que é executado *antes* do construtor, para *cada* nova instância da classe. É útil para inicialização comum a todos os construtores.
• **Ordem de Execução na Criação de um Objeto (com Herança):**
1. **Inicialização de membros estáticos** da superclasse (se houver, ocorre apenas uma vez quando a classe é carregada).
2. **Inicialização de membros estáticos** da subclasse (se houver, ocorre apenas uma vez quando a classe é carregada).

3. **Blocos de inicialização de instância** da superclasse.
4. **Construtor da superclasse** (chamado implicitamente ou explicitamente via `super()`).
5. **Blocos de inicialização de instância** da subclasse.
6. **Construtor da subclasse**.

 # 4. Membros Estáticos (`static`)

• Membros (atributos ou métodos) declarados com `static` pertencem à **classe**, não a uma instância específica do objeto.
• São acessados usando o nome da classe (ex: `CAx.op3(x)`).
• Métodos estáticos não podem acessar atributos ou métodos de instância diretamente (pois não há um objeto `this` associado a eles).
• Como mencionado, métodos estáticos não podem ser sobrescritos, apenas ocultados.

 # 5. Tratamento de Exceções (`try-catch-finally`)

• **Exceção:** Um evento que interrompe o fluxo normal de execução de um programa.
• **`try` bloco:** Contém o código que pode gerar uma exceção.
• **`catch` bloco:** Captura e trata uma exceção específica que foi lançada no bloco `try`.
• **`finally` bloco:** Contém código que **sempre será executado**, independentemente de uma exceção ter ocorrido ou não, ou se um `return` foi executado no `try` ou `catch`. É ideal para liberar recursos (fechar arquivos, conexões).
• **`throws` palavra-chave:** Usada na assinatura de um método para indicar que o método pode lançar uma exceção específica. O chamador do método deve lidar com essa exceção (usando `try-catch`) ou declará-la também com `throws`.
• **`throw` palavra-chave:** Usada para lançar uma exceção explicitamente. Ex: `throw new Exception();`.

 # 6. Comparação de Strings

• **`==` operador:** Compara as **referências** de objetos. Para strings, ele verifica se as duas variáveis apontam para o *mesmo objeto* na memória.
• **`.equals()` método:** Compara o **conteúdo** das strings. É a forma correta de verificar se duas strings têm os mesmos caracteres.
* Ex: `if (x == null)` é uma verificação válida para saber se a referência `x` não aponta para nenhum objeto. No entanto, `if (x == "abc")` é geralmente incorreto para comparar conteúdo; use `x.equals("abc")`.

Ao analisar as questões, preste atenção especial à ordem de execução de construtores e blocos de inicialização, como o polimorfismo afeta a chamada de métodos de instância versus estáticos, e o fluxo de controle em blocos `try-catch-finally`.

Questões de Provas Anteriores

Fonte: escriturario_agente_de_tecnologia.pdf, Página: 19

pcimarkpci MjgwNDowMTRkOjE0YTU6OTI1ODozOGQ2OjNhMGM6NTM0MzplZmI1:U3VuLCAyNyBKdWwgMjAyNSAyMzo0NzozMSAtMDMwMA==
www.pciconcursos.com.br
19
BANCO DO BRASIL
AGENTE DE TECNOLOGIA - Microrregião 158 -TIGABARITO 1
64
Sejam as seguintes classes Java, que ocupam arquivos separados:

```
public class CAx {
protected int a;
protected int b;

public CAx() {
a*=2;
b*=3;
}

{
a=1;
b=2;
}

public int op1(int x) {
return op2(x)+op3(x)+b;
}

public int op2(int x) {
return x+a;
}

public static int op3(int x) {
return x*2;
}
}
public class CBy extends CAx {
protected int a;

public CBy() {
```

```
a+=3;
b+=3;
}

public int op2(int x) {
return x-a;
}

public static int op3(int x) {
return x*3;
}
}
public class Main {
public static void main(String[] args) {
CAx o=new CBy();

System.out.println(o.op1(2));
}
}
```

O que será exibido no console quando o método main for executado?

(A) 10 (B) 12 (C) 14 (D) 18 (E) 20

Fonte: escriturario_agente_de_tecnologia.pdf, Página: 20

pcimarkpci MjgwNDowMTRkOjE0YTU6OTI1ODozOGQ2OjNhMGM6NTM0MzplZmI1:U3VuLCAyNyBKdWwgMjAyNSAyMzo0NzozMSAtMDMwMA==

BANCO DO BRASIL
AGENTE DE TECNOLOGIA - Microrregião 158 -TI GABARITO 1
65
Considere as seguintes classes Java, que ocupam arquivos separados:

```
public class Pa {
String x,y,z;
String r="vazio";

public Pa(String s1,String s2, String s3) throws Exception {
x=s1;
y=s2;
z=s3;
```

```
try {
if(x==null || y==null || z==null)
throw new Exception();
}
catch(Exception e) {
z= "a";
throw e;
}
finally {
if(x==null)
x= "***";
if(y==null)
y= "***";
if(z==null)
z= "***";
}
}

public String get() {
return r;
}
}
public class Qb extends Pa {

public Qb(String s1,String s2, String s3) throws Exception {
super(s1,s2,s3);
r=x+y+z;
}
}
public class Main {
public static void main(String[] args) {
Pa o=null;

try {
o=new Qb( "a"," ","c");
}
catch (Exception e) {
System.out.print( “***Erro***“);
}
finally {
if(o!=null)
System.out.print(o.get());
}
```

```
}
}
```