POO

**Por que usamos "public static void main (String [] args)" em Java?**

Em Java, a classe que contém o método "main" é considerada a classe principal do programa e é o ponto de entrada para a execução do programa. O método "main" é o primeiro método a ser executado quando o programa é iniciado e, portanto, é necessário que ele seja definido corretamente.

A palavra-chave "public" é usada para indicar que o método "main" pode ser acessado por qualquer classe que tenha acesso à classe principal. No entanto, como o método "main" é o ponto de entrada do programa e geralmente é usado apenas dentro da classe principal, ele pode ser definido como "private" para limitar o acesso a esse método.

A palavra-chave "static" é usada para indicar que o método "main" pertence à classe, não a uma instância específica da classe. Isso significa que o método "main" pode ser chamado sem criar um objeto da classe. Isso é necessário porque o método "main" é chamado pelo sistema de execução do programa, antes que qualquer objeto seja criado.

A palavra-chave "void" é usada para indicar que o método "main" não retorna nenhum valor. Isso ocorre porque o método "main" é apenas o ponto de entrada do programa e não é projetado para retornar valores.

O parâmetro "String[] args" é utilizado no método "main" em Java para receber argumentos da linha de comando. Quando um programa Java é executado a partir da linha de comando, é possível passar argumentos adicionais para o programa. Esses argumentos são passados como uma matriz de strings (array de strings) para o método "main".

Por exemplo, se você executar um programa Java chamado "MeuPrograma" a partir da linha de comando com o seguinte comando:

java **MeuPrograma** argumento1 argumento2 argumento3

Os argumentos **"argumento1", "argumento2" e "argumento3"** serão passados para o método "**main**" como uma **matriz de strings** com três elementos:

String[] args = {"argumento1", "argumento2", "argumento3"};

Esses argumentos podem ser úteis para fornecer informações adicionais ao programa durante a execução, como a configuração do programa ou a especificação de um arquivo de entrada.

Portanto, o parâmetro "String[] args" no método "main" é utilizado para receber argumentos da linha de comando e permitir que o programa seja executado com diferentes configurações ou opções.

POO

**Abstração de Dados**

Dados e lógica da aplicação são organizados em Classes, que podem ser instanciados para criar objetos.

**Classe**

Abstrata, possui propriedades e tem comportamentos.

**Objeto**

Concreto, possui valores e faz o comportamento

Importância: Programação estruturada: estruturas de controle/iterativas e funções.

POO => Modularizou a lógica, dados e conceitos. Nasceu, as Classes.

A Programação Orientada a Objetos modularizou a lógica, dados e conceitos de uma aplicação na forma de funções!

Verdadeiro

**Falso x**

A Programação Orientada a Objetos modularizou a lógica, dados e conceitos de uma aplicação na forma de classes!

Caracterizar e Identificar responsabilidades de uma classe, serve para facilitar a identificação de atributos e comportamentos de classes.

**Entidades**

**Entidades** colaboram para criar a funcionalidade da aplicação, elas são **chamadas de Classes**; Ex.: Animal, Produto, Usuário, Cliente. A classe é uma abstração, que possui características, ex: Cadeira.

O **objeto** é uma **instância concreta da classe**.

**Classe Cadeira** => cria/instancia => **objetos cadeiras**.

Os objetos são únicos.

Qual dos objetos são instâncias da classe Cadeira?

Cadeira de rodas

Cadeira de bebê

Banquinho

Duas cadeiras visualmente iguais são objetos diferentes?

Sim

Não

**Comportamentos e Estado das Classes**

O estado é representado pelo conjunto de valores referentes aos atributos enquanto os métodos representam o comportamento de um objeto. Objetos podem conter Objetos, ou seja, os atributos de um objeto podem ser objetos da mesma classe ou de classes distintas.

As características de um objeto da classe são chamadas de atributos.

A classe também pode realizar ações, **comportamentos**. Uma ação pode inclusive mudar valores de atributos de um objeto.

As ações que um objeto da classe pode realizar são representadas por métodos.

O **estado** **de uma classe** pode ser representado por valores de atributos de um objeto?

atributos

O **comportamento** de uma classe pode ser representado por?

métodos

**Classe**: **Estado**(atributos) e **comportamento**(métodos).

|  |  |
| --- | --- |
| **Classes** | **Objetos** |
| Abstrato | Concreto |
| Tem atributos | Possui valores |
| Tem ações | Executa ações |

1. Quais são as peças ou entidades que usamos em um software orientado a objetos?

**Classes e Objetos**

Em orientação objetos consideramos dois tipos de entidades: classes, que são entidades abstratas, e objetos, que são instâncias concretas de classes.

2.

Escolha a alternativa correta:

Classe e objeto são instâncias concretas

Classe e objetos são abstrações

Classe é uma instância concreta e objeto é uma abstração

Classe é uma abstração e objeto é uma instância concreta

Classe é uma abstração e objeto é uma instância concreta de uma dada classe. Por exemplo, a classe cadeira pode ter como objeto, apenas para exemplificar, uma cadeira de escritório e uma cadeira de bebê!

3.

Um banquinho de sentar não pode ser uma instância da classe Cadeira porque lhe falta um atributo característico de cadeira. Qual é esse atributo?

Encosto

A classe Cadeira tem três atributos que a caracterizam: encosto, assento e pés. Um banquinho de sentar tem dois atributos: assento e pés. Portanto, o atributo que lhe falta é o encosto!

4.

Objetos são criados a partir de classes

Verdadeiro

Falso

5.

Um objeto descreve o tipo da classe!

Falso

Verdadeiro

Uma classe é que descreve o tipo do objeto!

Uma classe é que descreve o tipo do objeto!

6.

Objetos representam instâncias individuais de uma classe

Verdadeiro

Falso

7. Dada uma classe Carro:

Só podemos criar um objeto dessa classe

Podemos criar tantos objetos dessa classe quanto desejarmos

Não podemos criar objeto algum

Só podemos criar um número fixo, digamos, 10, de objetos dessa classe.

8.

A afirmação é verdadeira (Selecione quantas forem verdadeiras):

Um programa Java em execução é composto de objetos das classes, fonte Java correspondente, comunicando-se pela troca de mensagens ou invocação de métodos

Um programa fonte Java é composto de classes Java

Um programa fonte Java é composto de objetos de classes Java

Um programa Java em execução é composto de classes, fonte Java correspondente, comunicando-se pela troca de mensagens ou invocação de métodos

1.

Uma abstração em um software orientado a objetos é representado com o conceito de \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_!

Função

Objeto

Classe

Herança

2.

Marque a alternativa correta sobre classe e objeto

Classe é uma instância concreta e objeto é uma abstração

Classe e objetos são abstrações

Classe e objeto são instâncias concretas

Classe é uma abstração e objeto é uma instância concreta

3.

Como é representado o estado e o comportamento de uma classe

O estado é representado por atributos e o comportamento por métodos

O estado é representado por métodos e o comportamento por atributos

Ambos são representados por atributos

Ambos são representados por métodos

4.

Marque as alternativas que apresentam corretamente uma classe e seus objetos

Fruta: Penca de banana, Cesta de Maçãs

Carro: Esportivo, Utilitário, Luxuoso

Gato: Garfield, Frajola, Tom

Pais: Brasil, Estados Unidos, Japão

5.

Dada a classe Cadeira, um conceito abstrato, uma cadeira específica de escritório podem ser caracterizados como um exemplo concreto da classe Cadeira. Em orientação a objetos, qual é o nome que esse exemplar concreto recebe?

Objeto

**Herança**

Em um sistema podemos ver a mesma entidade de forma mais concreta ou mais abstrata.

Cliente é uma Pessoa.

As abstrações podem ser definidas em vários níveis, onde um pode herdar características de outro.

Abstrações mais gerais e mais específicas.

Ex.: transporte terrestre, abstrato => carro => SUV, concreto

Usado em funcionalidade que precisa da velocidade máxima e média, tempo de viagem.

Usado em funcionalidade que precisa alocar carga, SUV.

Diagrama

Descrição gerada automaticamente

**Interface** => Comportamento

**Classe** => Conceito/Definição

Utilizar herança para trabalhar com vários níveis de abstração.

Em Java, você pode criar duas classes que compartilham um comportamento semelhante, mesmo que não estejam relacionadas de outra forma, fazendo-as ambas \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ a mesma \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Escolha o par mais apropriado!

estenderem/classe

herdarem/classe

implementarem/interface XXX

implementarem/classe

herdarem/interface

estenderem/interface

Em Java, se você tiver duas classes que são ambas versões especializadas de uma classe mais geral, você pode ter ambas as classes \_\_\_\_\_\_\_\_\_\_\_\_ a mesma \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

implementando/interface

estendendo/superclasse XXX

estendendo/interface

implementando/superclasse

**Construtores**

São métodos especiais usados para criar objeto da classe. Com eles você pode parametrizar o objeto criado e inicializar variável.

Não precisa ter retorno, possui o mesmo nome da Classe, o **this** utilizado para referenciar elementos da classe. Para diferenciar o parâmetro do atributo, utiliza-se o **this**.

Toda classe tem pelo menos um construtor, mesmo quando não definimos nenhum explicitamente?

Sim

Não

Quando nenhum construtor é definido, um vazio e sem parâmetros é criado automaticamente.

É possível ter mais de um construtor com tipos de parâmetros diferentes.

**Nota: Não é possível ter dois construtores com a mesma assinatura, Carro(int).**

Carro(int potencia){ this.potencia = potencia; }

Carro(int velocidade){ this.velocidade = velocidade; }

Para termos mais de um construtor em uma classe, os parâmetros necessariamente precisam:

ter quantidades diferentes

ter tipos diferentes

Com tipos diferentes, mesmo com a mesma quantidade, é possível termos mais de um construtor.

**Encapsulamento**

A importância de esconder o comportamento da interface. Para interagir com alguma coisa, você não precisa saber como ela funciona internamente.

Consiste em esconder o comportamento interno das classes das classes externas.

Quando a interface é a mesma, fica mais fácil mudar o que está por trás.

Connection c = ConectionFactory.create();

c.open();

c.sendInformation(“test”);

c.close();

**Nota**: **DatabaseConnection; FileConnection; SocketConnection**

**Polimorfismo**

Poli, significa muitos e morfismo, significa formas.

Polimorfismo é um objeto poder assumir a forma de qualquer uma de suas abstrações.

Polimorfismo permite que uma nova classe seja utilizada por um método já existente.

Seja o seguinte trecho de código Java:

public interface IVegetariano{. . .}

public class Animal{. . .}

public class Coelho extends Animal implements IVegetariano{. . .}

Nós dizemos que a classe é polimórfica. Aponte, pelo número da linha de código [4 – 12], quais das seguintes atribuições são legais em Java:

public class Cliente {

    public static void main(String[] args) {

        Coelho coelho = new Coelho();

        Animal animal = coelho;

        Coelho coelho02 = new Animal( );

        IVegetariano vegetariano = new IVegetariano( );

        IVegetariano vegetariano02 = coelho;

        Object animal02 = coelho;

        IVegetariano vegetariano03 = new Animal( );

        animal = animal02;

        Animal animal03 = vegetariano;

    ...

    }

}

OO

* Conceitos de Classe
* Atributos e Responsabilidades
* Associação
* Multiplicidade
* Navegabilidade: direção das setas do relacionamento
* Agregação/composição/herança
* Diagramas: de Classes; Objetos

**Conceitos de Classe** – Representação – **Arquitetura do código**

|  |
| --- |
| **Nome da Classe** |
| Atributos |
| Operações ou métodos |

**Nota**: O relacionamento ou colaborações entre as classes é muito importante, representado pelo diagrama de classe.

|  |
| --- |
| **Pedido** |
| - data : **Date**  - status: **int** = 0  - itemPedido [0..\*]: **DetalhePedido**  - pagamento: **Pagamento** |
| - getAliquota( ): **int**  #getData( ): **Date**  #setData(data:Date ): **void**  #getStatus( ): **int**  #setStatus(status: int): **void**  +calcTax(alíquota: int ): **double**  +calcTotal( ): **double**  +calcTotalPeso( ): **double** |

Ex.:

**Atributos / variáveis de instância**

**Relacionamentos / Colaborações**

**Responsabilidades / Operações / Métodos**

**Métodos de Acesso:**

**getter/setter**

**Visibilidade:**

**- private**

**# protected**

**+ public**

**Representações**

|  |
| --- |
| **Pedido** |

|  |
| --- |
| **Pedido** |
|  |
| +calcTax(alíquota: int ): **double**  +calcTotal( ): **double**  +calcTotalPeso( ): **double**  ... |

|  |
| --- |
| **Pedido** |
| - data : **Date**  - status: **int** = 0  ... |

**Relacionamentos entre Classes:**

* **Associação – Tem 🡪**
* **Associação**
  + Agregação - **Parte**
  + Composição - **Possui**
* **Herança ou Generalização**
* **Dependência – Depende ---->**

**Multiplicidade: 1..1; 1..\*; \*..1; \*..\***

Conjunto de objetos que pode instanciar relativo a um tipo de associação.

|  |
| --- |
| **Cliente** |

|  |
| --- |
| **Pedido** |

0..\*

1

**faz**

1

**refere-se**

1..\*

|  |
| --- |
| **Pagamento** |

**0**..1 = **Opcional**

**1**..1 = **Obrigatório**

Diagrama

Descrição gerada automaticamente

Diagrama

Descrição gerada automaticamente

**Navegabilidade**

Tem a ver com o que a classe enxerga da outra.

Bidirecional:

Unidirecional: OU

|  |
| --- |
| **Pagamento** |

A classe **Pedido** enxerga a classe **Pagamento**.

|  |
| --- |
| **Pedido** |

**Associação Agregação**: Diamante aberto – **tem-um; é-parte-de => TEM e PERTENCE; CONCEITO DE PARTE**

Se o Carro for eliminado, os objetos Roda continuarão existindo.

Ex.: Classe **Carro** e Classe **Roda**

Imagem em branco e preto

Descrição gerada automaticamente com confiança média

**Associação Composição**: Diamante fechado – **tem-um; é-parte-de – POSSUI; CONCEITO DE TODO**

Se o Livro for eliminado, os objetos Capítulo serão eliminados também.

Ex.: Classe **Livro** e Classe **Capítulo**

Imagem em branco e preto

Descrição gerada automaticamente com confiança média

Na agregação, se um objeto agregador for eliminado, os objetos agregados serão eliminados também! Verdadeiro ou Falso?!? **Falso**

Na agregação, se um objeto agregador for eliminado, os objetos agregados continuarão a existir, pois eles já existiam antes de serem agregados, de modo que quando o objeto agregador for removido, isso não afetará os objetos agregados, pois a vida deles não dependia da vida do objeto agregador!

Na composição, se um objeto que compõe outros objetos for eliminado, os objetos da composição serão eliminados também! **Verdadeiro**

Como os objetos de uma composição são criados dentro do objeto que os compõem, a sua vida está atrelada à vida do objeto que os compõem; de modo que quando ele for eliminado, todos os objetos da composição serão eliminados juntos!

**Generalização / Especialização**

Responder à pergunta de relacionamento **é-um?** Técnico **é um tipo de empregado?** Engenheiro **é um tipo de empregado? etc**

Herança relaciona uma classe com a sua subclasse.

**Nota**: Em herança **classe Abstrata** – **representar em itálico a classe no diagrama** – não é permitido instanciar. Somente classes concretas podemos instanciar.

**Símbolo**: seta com triângulo vazado.

Diagrama

Descrição gerada automaticamente

**Classe Abstrata**

Diagrama

Descrição gerada automaticamente

**Classe Abstrata**

**Classe Concreta**

Pergunta 1

Diagrama

Descrição gerada automaticamente

A figura é um exemplo de:

Dependência

Composição

Associação

Agregação

Herança

Pergunta 2

Forma, Retângulo

Descrição gerada automaticamente

A figura é um exemplo de:

Associação e a classe A tem visibilidade da classe B, mas classe B não tem visibilidade da classe A

Herança e classe A herda da classe B

Dependência e classe A depende da classe B

Composição e a classe A tem a classe B como componente

Agregação e a classe A agrega a classe B

Pergunta 3

Texto

Descrição gerada automaticamente com confiança média

A figura é um exemplo de:

Dependência e a classe Estudante depende da classe Professor

Associação bidirecional: a classe Estudante tem visibilidade da classe Professor e a classe Professor tem visibilidade da classe Estudante

Associação unidirecional e a classe Estudante tem visibilidade da classe Professor, mas a classe Professor não tem visibilidade da classe Estudante

Herança e classe Estudante herda da classe Professor

Composição e a classe Estudante tem a classe Professor como componente

Agregação e a classe Estudante agrega a classe Professor

4.

Pergunta 4

Texto

Descrição gerada automaticamente com confiança baixa

A figura é um exemplo de:

Agregação e a classe Universidade agrega a classe Estudante

Herança e classe Estudante herda da classe Universidade

Associação e a classe Universidade tem visibilidade da classe Estudante, mas classe Estudante não tem visibilidade da classe Universidade

Composição e a classe Universidade tem a classe Estudante como componente

Dependência e classe Universidade depende da classe Estudante

Pergunta 5

Que relacionamento é apropriado entre os seguintes pares de classes: herança ("é–um"), associação, agregação ("tem–um") e composição ("tem–um"):

Carro – Porta

Caminhão – Veículo

Tráfego – Sinal de Trânsito

Sinal de Trânsito – Cor

Agregação – Herança – Associação – Composição

Composição – Herança – Associação – Agregação

Associação – Composição – Agregação – Herança

Agregação – Associação – Herança – Composição

Pergunta 6

Na agregação, se um objeto agregador for eliminado, os objetos agregados são eliminados também.

Verdadeiro

Falso

Pergunta 7

Na composição, se um objeto que tem os objetos componentes for eliminado, os objetos componentes são eliminados também? Por quê?

Sim, porque, ao serem criados dentro do objeto como um componente, eles não tinham vida própria antes do objeto começar a criá-los.

Sim, porque, ao serem criados dentro do objeto como um componente, eles tinham vida própria antes do objeto começar a criá-los.

Não, porque, ao serem criados dentro do objeto como um componente, eles não tinham vida própria antes do objeto começar a criá-los.

Não, porque, ao serem criados dentro do objeto como um componente, eles tinham vida própria antes do objeto começar a criá-los.

Pergunta 8

Seja o trecho de código Java abaixo:

public class Faculdade {

   public Faculdade (Departamento departamento){

      \_departamento = departamento;

   }

. . .

   Departamento \_departamento;

}

Considerando o relacionamento Faculdade–Departamento como sendo "tem–um" e a forma como esse relacionamento foi implementado no trecho acima, diga se a associação é uma agregação ou uma composição e por quê?

Agregação, porque, ao passar o objeto Departamento relacionado pelo construtor, significa que ele não tinha vida própria antes de ser associado ao objeto Faculdade.

Composição, porque, ao passar o objeto Departamento relacionado pelo construtor, significa que ele não tinha vida própria antes de ser associado ao objeto Faculdade.

Composição, porque, ao passar o objeto Departamento relacionado pelo construtor, significa que ele tinha vida própria antes de ser associado ao objeto Faculdade.

Agregação, porque, ao passar o objeto Departamento relacionado pelo construtor, significa que ele tinha vida própria antes de ser associado ao objeto Faculdade.

**Colaboração, dependência, classe cliente e classe servidora.**

**Dependência entre classes - - - ->**

classe cliente depende da classe servidora

Quando um objeto, por exemplo, Biblioteca, não pode existir sem outro objeto, no caso, Livro, então o objeto Biblioteca depende do objeto Livro, assim como depende da Classe Usuário, clientes da Biblioteca.

Uma classe é cliente quando ela necessita ou depende da colaboração de uma classe servidora, que tem a responsabilidade ou método que ela requer.

**Quando o objeto cliente utiliza o objeto Servidor como:**

* Argumento de construtor/método
  + public Empresa ( **Mercadoria** mercadoria ) {...} // argumento
  + private void **asociaMercadoria** ( **Mercadoria** mercadoria ) {...} // parâmetro
* Tipo de variável de instância ou local
  + private **Mercadoria** mercadoria;
* Tipo de retorno
  + public **Mercadoria** getMercadoria ( ) { **return** mercadoria; }

Quais são as classes servidoras de uma dada classe em termos de colaboração / dependência?

* Outra classe
* Superclasse - **herança**
* A própria classe

Os quatro conceitos formam a base dos Padrões de Projeto!

* **abstração de dados** – ar – essencial – definição
* **herança** – água profundidade
* **encapsulamento** – terra – barreiras
* **polimorfismo** – fogo – várias formas – transformação

**Responsabilidade do tipo Faz: public; private**

Responsabilidade do tipo **private**, não pode colaborar com instância de subclasse. Ou seja, método privado não é herdado.

O método privado da superclasse não é visível a um objeto da subclasse; portanto, tanto com "**this**", quanto com "**super**", não irá funcionar!

**Contrato de uma Classe**

**Contrato** é o conjunto de **serviços/comportamentos** de um objeto.

Pode ser requisitado, **para colaborar com**, por outros objetos.

Contrato também é conhecido por:

* Conjunto de **responsabilidades públicas**
* Protocolo
* Interface padrão
* Interface externa
* Interface da classe

**Padrões de Projeto de Acordo com o GoF**

**Sobre Conceitos Essenciais para Padrões de Projeto**

**1.**

A Programação Orientada a Objetos modulariza a lógica, dados e conceitos de uma aplicação na forma de classes!

Verdadeiro

Falso

**2.**

I. Conceito de overloading ou sobrecarga de métodos: definem-se vários métodos de mesmo nome e com assinaturas diferentes

II. Conceito de overriding ou sobreposição: altera-se o comportamento do método na subclasse, mantendo a mesma assinatura, mas com funcionalidade diferente

Somente II está correto

Somente I está correto

I e II estão incorretos

I e II estão corretos

**3.**

Nós usamos o termo encapsulamento para significar uma divisão um tanto frouxa entre a visão interna e visão externa de uma classe

F

V

Nós usamos o termo encapsulamento para significar uma divisão **precisa** entre a visão interna e visão externa de uma classe

**4.**

Uma Interface pode ser associada a comportamento, enquanto Classe a conceito!

F

V

**5.**

Olhe para o diagrama de classes abaixo e decida qual das seguintes linhas de código seria legal em um programa Java onde essas classes tenham sido implementadas:

**[Obs. 1: Pode haver mais de uma resposta correta!**

Diagrama

Descrição gerada automaticamente

1

Revista revista = new RevistaDigital( );

Está OK, porque RevistaDigital é uma subclasse de Revista

RevistaDigital revistaDigital = new Revista( );

Está ilegal, porque Revista é uma superclasse de RevistaDigital

Publicação publicação = new Livro( );

Aqui estamos definindo uma variável publicação do tipo geral de Publicação; estamos então invocando o construtor para a classe Livro e atribuindo o resultado a publicação; isso está OK, porque Livro é uma subclasse de Publicação. Ou seja, um livro é uma publicação.

Publicação publicação = new Publicação( );

Está ilegal, porque a Publicação é uma classe abstrata e, portanto, não poderia ser instanciada.

Publicação p = new RevistaDigital( );

Está OK, porque RevistaDigital é uma subclasse de Revista, que, por sua vez, é uma subclasse de Publicação. Ou seja, RevistaDigital é uma subclasse indireta de Publicação.

1

**6.**

Observe o diagrama de classes abaixo e, supondo que a classe Publicação não seja abstrata, decida quais dos pares de operações mostrados são legais em um programa Java onde essas classes tenham sido implementadas: **[Obs. 1: Pode haver mais de uma resposta correta!**

Diagrama

Descrição gerada automaticamente

Publicação publicação = new Publicação ( );

publicação.recebeNovaEdição( );

Não deve ser selecionado

Ilegal – O método recebeNovaEdição() não existe em objeto de Publicação, apenas em objeto de Revista.

Publicação publicação = new Revista( );

publicação.recebeNovaEdição( );

Não deve ser selecionado

Ilegal – Embora nós possamos enviar mensagem recebeNovaEdição( ) a um objeto Revista, neste caso, o compilador não sabe que 'publicação' dinamicamente faz referência a um objeto Revista; o compilador só sabe que 'publicação' estaticamente é um objeto Publicação.

Publicação publicação = new Revista( );

publicação.vendeCópia( );

Legal – Revista é um tipo de Publicação e, portanto, você pode atribuir um objeto de tipo Revista a uma variável de tipo Publicação. Você sempre pode substituir subtipos onde um supertipo é solicitado: Liskov Substitution Principle – LSP, que é o "L" do SOLID. Também você pode enviar mensagem vendeCópia para objeto de classe Publicação.

A variável "publicação" passa dinamicamente a referenciar um objeto da classe Revista, mas isso é irrelevante neste caso, na medida em que o compilador sabe que "publicação" é estaticamente apenas um objeto da classe Publicação.

Em tempo de execução, a máquina virtual do Java verifica se o objeto de Revista denotado pela variável "publicação" tem o método vendeCópia( ) se tivesse, esse método é que seria executado. Como Revista não tem, a máquina virtual verifica se a superclasse imediata, no caso Publicação, tem o método vendeCópia( ); como tem, é esse método de Publicação que é executado!

Revista revista = new Revista(...);

revista.recebeNovaEdição( );

Legal – A variável 'revista' é um objeto Revista e nós podemos enviar a mensagem recebeNovaEdição( ) a objetos da classe Revista.

Publicação publicação = new Publicação( );

publicação.vendeCópia( );

Legal – Você pode enviar mensagem vendeCópia para objeto de classe Publicação

**7.**

Veja o diagrama de classes abaixo e, observando que Estudante é uma classe abstrata, decida qual dos dois seguintes códigos são legais em Java:

Diagrama

Descrição gerada automaticamente

Item I

Estudante estudante = new Estudante( );

Professor professor = new Professor( );

professor.ajuda(estudante);

Item II

Estudante estudante = new EstudanteTempoIntegral( );

Professor professor = new Professor( );

professor.ajuda(estudante);

Escolha a afirmação verdadeira:

Itens I e II não são verdadeiros

Itens I e II são verdadeiros

Apenas item II é verdadeiro

Apenas item I é verdadeiro

**8.**

Deve-se tomar muito cuidado com o termo 'interface', já que na programação Java ele tem pelo menos três significados:

I. Os membros públicos de uma classe

II. A "interface do usuário" de um programa, muitas vezes uma "interface gráfica do usuário"

III. Uma construção Java específica que serve para descrever o comportamento que outras classes implementam

Escolha a afirmação verdadeira:

Apenas III é verdadeira

I, II e III não são verdadeiras

Apenas I e II são verdadeiras

Apenas I e III são verdadeiras

Apenas II e III são verdadeiras

I, II e III são verdadeiras

Apenas I é verdadeira

Apenas II é verdadeira

O item I, II e III válidos. EstudanteTempoParcial é um tipo de Estudante e pode ser atribuído a variável 'estudante' do tipo Estudante. E 'estudante' pode ser passada como um parâmetro para professor.ajuda (estudante).

**9.**

Escolha as alternativas corretas, se houver mais de uma!

O polimorfismo torna mais fácil estender nossos programas adicionando classes adicionais sem precisar alterar outras classes

O Java garante que o método apropriado para a classe real do objeto é invocado em tempo de execução

Podemos manipular objetos invocando operações definidas para a superclasse sem se preocupar com qual subclasse está envolvida em qualquer caso específico

Polimorfismo nos permite referir a objetos de acordo com uma superclasse em vez de sua classe real

**10.**

Uma variável polimórfica é uma variável que pode manter diferentes tipos de valores. A respeito disso, diga se a afirmação abaixo é verdadeira ou falsa:

* Java restringe os tipos de valores a objetos de subclasses do tipo declarado da variável polimórfica!

V

F

**11.**

Pergunta 11

Apenas um número limitado de classes pode implementar uma interface particular

F

V

**12.**

Ao escrever uma classe BoardGame final e concreta, você estendeu a classe GamePiece que requer que você escreva um método move( ) em BoardGame. Na superclasse GamePiece (que não tem superclasse a não ser Object e Object não define nenhum método move( )), o método move( ) deve então ser definido como um método \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

derivado

interface

protected

private X

public

abstract

concreto

final

**Criando Classes em Java**

Em Java quase tudo que você cria é uma classe.

public class Carro { ... }

As classes costumam ser organizadas em packages, pacotes.

**package** org.veiculo;

public class Carro { ... }

Para utilizar uma classe fora do seu package, deve importar todo o pacote, uma classe específica do pacote.

import java.util.\*;

import java.lang.Math;

As classes podem ser organizadas em...

packages

**A informação que uma classe precisa saber é armazenada nos atributos.**

public class Carro {

int potencia;

float velocidade;

}

Cada objeto da classe terá um valor diferente.

Todo atributo precisa ter um tipo. Um atributo pode ser de um **tipo primitivo** **ou de uma classe**.

**tipo** nome do atributo;

**Criando Objetos – operador new**

Carro **gol** = **new** Carro( ); // Carro, declara o tipo da variável.

Que operador utilizamos para instanciar uma classe?

new

**Acessando os atributos // os atributos podem ser acessados para escrita e leitura**

**gol**.potencia = 5;

float v = **gol**.velocidade;

**Comportamento – os métodos definem o comportamento da Classe.**

public **class** Carro {

void acelerar( ){ velocidade += potencia; } // manipula o estado da Classe.

void frear( ) { velocidade \*= 0.5; }

int getVelocidade( ) {return velocidade;}

void imprimir( ) { System.out.println(“Velocidade do Carro:” + getVelocidade( ) + “km/h”); }

}

**Retorno:** void = não retorna nada; pode ter ou não parâmetros.

**Executando métodos.** O **método** será executado no objeto utilizado para a chamada.

**gol**.acelerar( );

**gol**.frear( );

Na classe Carro o que são "potencia", "velocidade"?

Tipos

Funções

Pacotes

Variáveis

Atributos

Métodos

As modificações feitas numa instância irão afetar numa outra instância?

Sim

Não

* POO

**1.**

É um método especial de uma classe Java que tem o mesmo nome da classe, cuja execução se dá imediatamente após a instanciação de um objeto da classe, com o objetivo de alocar memória e iniciar as suas variáveis de instância. Como é chamado esse método especial?

**Construtor**

**2.**

Uma classe que tem um método abstrato não precisa ser declarada como abstrata. Verdadeiro/V ou Falso/F?

F

Uma classe que tem um método abstrato precisa ser declarada como abstrata.

**3.**

Uma classe abstrata pode não ter nenhum método abstrato.

Falso

Verdadeiro

**4.**

O código abaixo compilará corretamente **(indique apenas e exatamente V ou F).** [Supondo que o trecho representado por "..." compila corretamente]

public abstract class X {...}

public class Teste{

   public static void main (String[] args) {

      X x = new X();

   }

}

F.

Classes abstratas não podem ser instanciadas, de modo que “new X( )” não compilará!

**5.**

O código abaixo compilará corretamente:

class Teste{

   public void metodo (int i) {...}

   protected void metodo (double x) {...}

}

Falso

Verdadeiro

Nesse caso, o overloading ou sobrecarga é legítimo, pois os argumentos dos dois métodos com mesmo nome são de tipos diferentes!

**6.**

O código abaixo compilará corretamente **(indique apenas e exatamente V – F)** [Supondo que o trecho representado por "..." compila corretamente]

class Teste{

   public int metodo ( ) {...}

   protected double metodo ( ) {...}

}

**F.**

* Num caso de overloading ou sobrecarga legítimo, um método pode ter o mesmo nome de outro, desde que os seus argumentos sejam diferentes, quanto à quantidade de argumentos ou quanto aos tipos dos argumentos se a quantidade for a mesma.
* Neste último caso, a ordem dos argumentos também importa: (int, double) é diferente de (double, int).
* Formalmente, dizemos que os métodos do exemplo têm o mesmo nome e mesma assinatura, o que é inaceitável em Java; seria aceitável se tivessem assinaturas diferentes quanto ao nome do método ou tipos e ordem de parâmetros!

**7.**

É um mecanismo existente no paradigma orientado a objetos que permite a reutilização da estrutura e do comportamento de uma classe ao se definir novas classes; é conhecido também como relacionamento "é um"; a classe que herda o comportamento é chamada de subclasse e a que definiu o comportamento, superclasse. Qual é o nome desse mecanismo?

Herança

Método

Polimorfismo

Responsabilidade

**8.**

Apresente como é em Java a assinatura do método correspondente à primeira mensagem que aparece no trecho do exemplo abaixo **[se você vislumbrar mais de uma assinatura possível, apresente apenas uma delas!]**[A assinatura de método em Java tem um formato que não segue a sintaxe do Java; a sintaxe é a de assinatura de método, que poderá ser usada ao se avaliar programas em C# ou Python, por exemplo!]:

. . .

Ponto ponto1 = new Ponto( );

Ponto ponto3 = new Ponto( );

. . .

if (ponto1.igual(ponto3))

      ponto3.mover(5,10);

. . .

O tipo de retorno e os nomes dos parâmetros não fazem parte da assinatura pro Java.

igual(Ponto)

**9.**

Os dados/operações de um objeto que são visíveis externamente compreendem seu/sua?

Interface

Sobrecarga

Sobreposição

Método

**10.**

O princípio de \_\_\_\_\_\_\_\_\_\_\_\_ permite que objetos que pertencem a diferentes classes respondam de forma distinta a mensagens idênticas.

**Polimorfismo**

**11.**

A principal diretriz convencionada para atributos de objetos é que todas as variáveis de instância devem ser declaradas \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Protegidas

Públicas

Privada