POO

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**

**Visibilidade:**

**- private**

**# protected**

**+ public**

**Responsabilidades / Operações / Métodos**

**Métodos de Acesso:**

**getter/setter**

**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** |

**Navegabilidade**

Bidirecional:

Unidirecional: OU

**Associação Agregação**: Diamante aberto – **tem-um – tem e pertence; PARTE**

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

Ex.: Classe Carro e Classe Roda

**Associação Composição**: Diamante fechado – **é-parte-de – tem e faz parte; POSSUI**

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

Ex.: Classe Livro e Classe Capítulo

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!

Herança / Generalização / Especialização / Relacionamento é um

Herança relaciona uma classe com a sua subclasse.

**Símbolo**: seta com triângulo vazado.

Como reflexão final, relembre mentalmente o que você viu na aula e, em seguida, liste 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

**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

Correto

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

**Encapsulamento**

Esconder 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

Como reflexão final, relembre mentalmente o que você viu na aula e escreva o que você entendeu por Encapsulamento!

Encapsulamento, a importância de esconder seu comportamento independente da interface.

**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

**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;

    ...

    }

}

**Sobre Conceitos Essenciais para Padrões de Projeto**

Prova de POO

[**Ir para o conteúdo principal**](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/quiz/nXXvD/sobre-conceitos-essenciais-para-padroes-de-projeto/attempt#main)

Parte superior do formulário

PESQUISAR NO CURSO

Pesquisar

Parte inferior do formulário

* Português (Brasil)
* **H**

Heleno Cardoso da Silva Filho

Ocultar menu

* **Introdução e Visão Geral do Curso**
* **Base Conceitual de Padrões de Projeto**

**[Vídeo:](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/oK5NG/orientacao-a-objetos-e-padroes)**[VídeoOrientação a Objetos e Padrões](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/oK5NG/orientacao-a-objetos-e-padroes)

[. Duration: 5 minutes5 min](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/oK5NG/orientacao-a-objetos-e-padroes)

**[Vídeo:](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/dbQBb/abstracao-de-dados)**[VídeoAbstração de Dados](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/dbQBb/abstracao-de-dados)

[. Duration: 9 minutes9 min](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/dbQBb/abstracao-de-dados)

**[Vídeo:](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/syAQX/encapsulamento)**[VídeoEncapsulamento](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/syAQX/encapsulamento)

[. Duration: 6 minutes6 min](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/syAQX/encapsulamento)

**[Vídeo:](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/vJLbI/heranca)**[VídeoHerança](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/vJLbI/heranca)

[. Duration: 8 minutes8 min](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/vJLbI/heranca)

**[Vídeo:](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/cXrgN/polimorfismo)**[VídeoPolimorfismo](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/cXrgN/polimorfismo)

[. Duration: 7 minutes7 min](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/lecture/cXrgN/polimorfismo)

**[Teste para praticar:](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/quiz/nXXvD/sobre-conceitos-essenciais-para-padroes-de-projeto)**[Sobre Conceitos Essenciais para Padrões de Projeto](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/quiz/nXXvD/sobre-conceitos-essenciais-para-padroes-de-projeto)

[. Duration: 30 minutes30 min](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/quiz/nXXvD/sobre-conceitos-essenciais-para-padroes-de-projeto)

* **Introdução a Padrões de Projeto**
* **Criação de Objetos – Parte 1**
* **Classificação dos Padrões de Projeto desta Semana de Acordo com o GoF**
* **Graded Quiz**

1. [Semana 1](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/home/week/1)
2. Sobre Conceitos Essenciais para Padrões de Projeto

**Sobre Conceitos Essenciais para Padrões de Projeto**

**1.**

Pergunta 1

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

Verdadeiro

Falso

**2.**

Pergunta 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.**

Pergunta 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.**

Pergunta 4

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

F

V

**5.**

Pergunta 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!**

**Obs. 2: Não vale rodar programas para testar os trechos de código antes de fazer este teste; após o teste, recomenda-se fortemente testar em programas, caso não tenha respondido corretamente!]**

Diagrama

Descrição gerada automaticamente

1

Revista revista = new RevistaDigital( );

Correto

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( );

Correto

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( );

Correto

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.**

Pergunta 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!**

**Obs. 2: Não vale rodar programas para testar os trechos de código antes de fazer este teste; após o teste, recomenda-se fortemente testar em programas, caso não tenha respondido corretamente!]**

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( );

Correto

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( );

Correto

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( );

Correto

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

**7.**

Pergunta 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.**

Pergunta 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

Incorreto

O item II é válido. 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.**

Pergunta 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.**

Pergunta 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.**

Pergunta 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

Padrões de Projetos com JAVA

* **Criação**
* **Estrutural**
* **Comportamental**

Vamos aproveitar para exemplificar o uso da classe ContaEstacionamento e da interface Calculo num cliente, bem como das classes que implementam a interface Calculo (CalculoHoraSimples, CalculoHoraTolerancia e CalculoDiaria), para entender como elas funcionam em conjunto:

. . .

ContaEstacionamento conta = new ContaEstacionamento( );

CalculoDiaria tipoCalculo = new CalculoDiaria(20);

// vamos injetar a dependência da forma de cálculo no objeto

// 'conta', onde "20" é o valor da diária numa dada época

conta.setCalculo(tipoCalculo);

double valor = conta.valorConta( ); // delega cálculo para

// objeto 'tipoCalculo'

System.out.println("Valor da Conta = " + valor);

. . .

O método valorConta( ) da classe ContaEstacionamento, calcula o valor delegando a lógica para o objeto da classe que implementa a interface Calculo; no exemplo existem 3 classes que implementam a interface Calculo .

Para que esse esquema todo funcione, quando se faz “conta.setCalculo(tipoCalculo);”, internamente significa que internamente à classe ContaEstacionamento atribui-se o objeto “tipoCalculo” a uma variável de instância da classe ContaEstacionamento cujo tipo seja um dos seguintes:

classe CalculoDiaria

Nem classe nem interface

classe ContaEstacionamento

classe CalculoHoraSimples

interface Calculo

Correto

O certo é interface Calculo, não classe Calculo!

No slide a seguir que mostra a classe ContaEstacionamento, está escrito o seguinte em vermelho na parte inferior do slide:

"O método que calcula o valor agora delega a lógica para o objeto da classe que implementa a **classe** Calculo"

O certo é dizer que a classe CalculoDiaria implementa a **interface Calculo**; ou seja, **Calculo** não é classe e sim uma **interface**!

|  |
| --- |
| <<**Interface**>>  **Calculo** |
| +valorConta() |

|  |
| --- |
| **ContaEstacionamento** |
| #veiculo  #inicio  #fim |
| +valorConta() |

|  |
| --- |
| **HoraSimples** |
| #valorHora |
| +calcular() |

|  |
| --- |
| **HoraTolerancia** |
| #valorHora |
| +valorConta() |

**public class ContaEstacionamento {**

private Calculo calculo;

private Veiculo veiculo;

private long inicio;

private long fim;

public **double** valorConta( ) {

**return** calculo.calcular(fim – inicio, veiculo ); **// O método que calculo o valor, agora delega a lógica para a classe Calculo.**

}

public **void** setCalculo(CalculoValor calculo){

**this**.calculo = calculo;

}

}

public class **CalculoDiaria** implements Calculo {

private double valorDiaria;

public **CalculoDiaria**( double valorDiaria ) {

**this**.valorDiaria = valorDiaria;

}

public double **calcular**( ) {

**return** valorDiaria \* Math.ceil( periodo / HORA );

}

}

**Nota**: Os padrões são soluções que podem ser utilizadas em vários contextos.

Padrão de projeto, é uma solução que pode ser utilizada para resolver um problema similar em diferentes contextos.

**Padrão de Projeto STRATEGY**

|  |
| --- |
| <<**Interface**>>  **Algoritmo** |
| +excutar() |

|  |
| --- |
| **Principal** |
| #atributos |
| +metodoPrincipal() |

**Contexto** **Problema**

|  |
| --- |
| **AlgoritmoConcreto2** |
| #atributos |
| +executar() |

**Solução 1** **Solução 2**

|  |
| --- |
| **AlgoritmoConcreto1** |
| #atributos |
| +executar() |

**Aplicabilidade 1 Aplicabilidade 2**

**CONSEQUÊNCIAS**

**Elementos dos Padrões de Projetos**

* Não é somente o diagrama.
* Contexto – a situação em que o problema ocorre – situação em que é aplicável.
  + Existe uma parte do comportamento de uma classe que deve executar diferentes algoritmos em diferentes situações.
* Problema – o que o padrão está tentando resolver
  + Encapsular uma família de algoritmos tornando-os intercambiáveis
* Solução – solução proposta, como e porque resolve o problema
  + Crie uma interface para representar o comportamento que varia, e componha a classe com uma implementação dela.
* Aplicabilidade – situação em que podem ser aplicadas este padrão
  + Situações em que o uso do padrão é adequado
  + Existem classes similares apenas com uma parte do comportamento diferente.
  + Um algoritmo pode ser alterado em tempo de execução
  + Muitas condicionais escolhem entre diferentes comportamentos
* Consequências – que podem ser negativas ou positivas, trade-off
  + **Positivas**: evita duplicação de código; remove condicionais; permite mudança em tempo de execução
  + **Negativas**: Aumenta número de classes; introduz complexidade

Alguns padrões podem ter uma estrutura parecida, mas serem aplicáveis em situações muito diferentes. Ex.: Camelo vs Dromedário.

**Erro comum**: Não se apegue a diagramas.

Os diagramas dos padrões Strategy e State, são muito parecidos:

Diagrama

Descrição gerada automaticamente

O que os diferenciam é o contexto de aplicação e o objeto de cada um deles.

Desafio: Após a apresentação do padrão de projeto State, explique por escrito as diferenças e similaridades entre os dois. Comece a comparação pelos objetivos de cada um e depois passe a analisar códigos Java que os apliquem. Pesquisar.

**Nota**: Padrão não é um pedaço de código, mas uma ideia que vai no seu cérebro.

**Padrões** possuem conhecimento de **design** utilizado em vários contextos ao redor do mundo.

Os padrões documentam boas práticas, assim como suas consequências e aplicabilidades.

Os **padrões** criam um **vocabulário** de **design** que permite os desenvolvedores referenciarem melhor as soluções utilizadas.

Os padrões é um conhecimento essencial para o desenvolvedor.

**Atividade:**

Como reflexão final, tente resumir as quatro afirmações discutidas, da seguinte forma:

"Padrão de Projeto = [use no máximo 3 palavras]

Por exemplo, para a primeira afirmação:

"Padrão de Projeto = ideia na cabeça!"

Qual seria o seu resumo para as outras 3 afirmações, incorporando, se for conveniente, as explicações discutidas!!!

* Padrão de projeto = soluções testadas e consolidadas.
* Padrão não é um pedaço de código = melhores práticas de uma solução
* Padrão cria um vocabulário de design = define um conceito
* Padrões documentam as boas práticas = formalizam uma solução.

Nota: Padrões: Strategy; Proxy; Observer, Adapter, etc

Dentre os princípios de **design orientado a objetos** apresentados nas alternativas abaixo, qual é o utilizado pelo **Strategy**?

* Abstração de Dados
* Herança
* Composição
* Encapsulamento

Atividade

**1.**

Pergunta 1

Os padrões GoF (Gang of Four) organizam um conjunto de padrões de projeto (*design patterns*) em três categorias: padrões de criação, padrões estruturais e padrões organizacionais.

V

F

Correto

Os 3 grupos de design patterns, segundo o GoF: padrões de criação, padrões estruturais e padrões **comportamentais**

**2.**

Pergunta 2

Sobre Padrões de Projeto:

I. São soluções reutilizáveis, testadas e gerais para problemas de software recorrentes

II. São independentes da linguagem de programação na qual o software será desenvolvido

III. Permitem que os algoritmos variem sempre independentemente dos clientes que os utilizam

Escolha a alternativa correta:

Apenas as opções II e III estão corretas

Apenas as opções I e III estão corretas

Apenas a opção II está correta

Apenas a opção I está correta

Nenhuma opção está correta

Apenas as opções I e II estão corretas

Apenas a opção III está correta

**3.**

Pergunta 3

O catálogo de padrões de projeto definido no GoF contém o seguinte:

24 padrões e está basicamente dividido em três categorias: padrões de criação, padrões espectrais e padrões comportamentais

21 padrões e está, basicamente, dividido em duas categorias: padrões de criação e padrões comportamentais

23 padrões e está, basicamente, dividido em duas categorias: padrões estruturais e padrões comportamentais

20 padrões e está, basicamente, dividido em duas categorias: padrões estruturais e padrões comportamentais

23 padrões e está, basicamente, dividido em três categorias: padrões de criação, padrões estruturais e padrões comportamentais

Correto

**4.**

Pergunta 4

* Descrição de conhecimento e experiência acumulados no desenvolvimento de software
* Solução bem-sucedida e testada para um problema de desenvolvimento de software
* Capta a experiência de solucionar problemas de desenvolvimento de software de uma maneira possível de ser reusada por outros

Indique qual conceito é caracterizado pelos três itens acima:

Linha de Produto de Software

Gerador de Programa

Framework

Padrão de Projetos

Desenvolvimento Baseado em Componentes

**5.**

Pergunta 5

Considere as seguintes afirmações sobre as vantagens do uso de padrões de software:

I. Padrões de projeto proporcionam um vocabulário comum de projeto, facilitando comunicação, documentação e aprendizado dos sistemas de software

II. Padrões de projeto auxiliam no desenvolvimento de software por meio da reutilização do projeto de soluções computacionais já testadas e aprovadas

III. Uma biblioteca de padrões pode ajudar a melhorar e padronizar o desenvolvimento de software

Quais são as assertivas corretas?

I, II e III

Somente I e III

Somente I

Nenhuma das assertivas!

Somente I e II

Somente II e III

Somente II

**6.**

Pergunta 6

O padrão Strategy destina-se a fornecer um meio para definir uma família de algoritmos e encapsular cada um como um objeto para uso permutável ou intercambiável

F

V

**Construtores**

Nem sempre o uso no new é a forma mais apropriada para se criar instâncias de classes. Pela falta de expressão de intenção do construtor.

Ex1.: Números Aleatórios com um número de dados; min >= 0; [min; max]

Gerar **inteiros aleatórios**, entre o min e o max, inclusive. **INTENÇÃO**

public **class** **GeradorIntAleatorio**{

private **final** int min;

private **final** int max;

**// valor aleatório ENTRE o min e o max, inclusive**

public **GeradorIntAleatorio**(int min, int max) {

**this**.min = **Math**.min(min, max);

**this**.max = **Math**.max(min, max);

}

public **int** **next**( ){

...

}

}

**GeradorIntAleatorio** gerAleat = **new** **GeradorIntAleatorio**(3,18); // **Falta clareza de INTENÇÃO**

int nrAleat = gerAleat.**next**( );

Ex2.: Gerar valor int aleatório. Onde o max deve ser Maior do que o min. **INTENÇÃO**

**// Gerar valores inteiros aleatórios MAIOR do que o valor min**

public **GeradorIntAleatorio**(int min) {

**this**.min = min;

**this**.max = **Integer**.*MAX\_VALUE*;

}

**GeradorIntAleatorio** gerAleat = **new** **GeradorIntAleatorio**(18); // **Falta clareza de INTENÇÃO**

int nrAleat = gerAleat.**next**( );

**Nota**: Será que a INTENÇÃO está explicita?

Ex3.: Gerar valor int aleatório. Onde o min deve ser Menor do que o valor max. **INTENÇÃO**

**// Gerar valores inteiros aleatórios MENOR do que o valor max**

static final int VALOR\_MINIMO = 0;

public **GeradorIntAleatorio**(int max) {

**this**.min = **VALOR\_MINIMO**;

**this**.max = **max**;

}

**GeradorIntAleatorio** gerAleat = **new** **GeradorIntAleatorio**(18); // **Falta clareza de INTENÇÃO**

int nrAleat = gerAleat.**next**( );

**Nota**: E não é permitido incluir mais de um construtor com a mesma assinatura.

public **GeradorIntAleatorio**(int) {...} // **assinatura**

Problemas com construtores

* Não expressam a intenção
* Impossível ter dois construtores com a mesma assinatura
* **new** X( ) , cria sempre um objeto novo da classe. X: Custo computacional
* Não consigo voltar o mesmo objeto outra vez
* Não consigo voltar um objeto da subclasse. New volta o objeto da classe e não objeto da subclasse
* Pode implicar em acoplamento concreto. Quando dentro de uma classe cria uma instância de outra classe

Solução: **Métodos Estáticos de Fábrica**. Static Factory Methods, é um padrão de criação desenvolvido especialmente para Java, mas pode ser utilizado em outras linguagens. Existem outros padrões: FactoryMethod; Abstract Factory Method.

Com este método estático é possível enviar mensagens para a classe porque ele é um método de classe. E o método de fábrica cria objetos de uma dada classe. Neste caso, definir o construtor como **PRIVATE**.

Uso de Static Factory Method no lugar de new para se criar instâncias de classes.

**Antes**

**// Gerar valores inteiros aleatórios ENTRE o min e o max, inclusive INTENÇÃO**

public **GeradorIntAleatorio**(int min, int max) {

**this**.min = **Math**.min(min, max);

**this**.max = **Math**.max(min, max);

}

**Depois**

**// Gerar valores inteiros aleatórios ENTRE o min e o max, inclusive**

**private** **GeradorIntAleatorio**(int min, int max) {

**this**.min = **Math**.min(min, max);

**this**.max = **Math**.max(min, max);

}

**// Gerar valores inteiros aleatórios ENTRE o min e o max, inclusive**

public static **GeradorIntAleatorio entre**(int min, int max) {

**return** new **GeradorIntAleatorio**(min, max);

}

Ou

public static **GeradorIntAleatorio getInstanceEntre**(int min, int max) {

**return** new **GeradorIntAleatorio**(min, max);

}

**GeradorIntAleatorio** gerAleat = **new** **GeradorIntAleatorio.getInstanceEntre**(3,18); //  **INTENÇÃO explicita**

int nrAleat = gerAleat.**next**( );

**Antes**

**// Gerar valores inteiros aleatórios MAIOR do que o valor min**

public **GeradorIntAleatorio**(int min) {

**this**.min = min;

**this**.max = **Integer**.*MAX\_VALUE*;

}

**Depois**

**// Gerar valores inteiros aleatórios MAIOR do que o valor min**

public static **GeradorIntAleatorio maiorQue**(int min) {

**return new GeradorIntAleatorio**(min, **Integer**.*MAX\_VALUE*);

}

**Antes**

**// Gerar valores inteiros aleatórios MENOR do que o valor max**

static final int VALOR\_MINIMO = 0;

public **GeradorIntAleatorio**(int max) {

**this**.min = **VALOR\_MINIMO**;

**this**.max = **max**;

}

**Depois**

**// Gerar valores inteiros aleatórios MENOR do que o valor max**

static final int VALOR\_MINIMO = 0;

public static **GeradorIntAleatorio menorQue**(int max) {

**return**.new **GeradorIntAleatorio**(**VALOR\_MINIMO**, max);

}

**GeradorIntAleatorio** gerAleat = **new** **GeradorIntAleatorio.menorQue**(18); //  **INTENÇÃO explicita**

int nrAleat = gerAleat.**next**( );

**GeradorIntAleatorio** gerAleat = **new** **GeradorIntAleatorio.maiorQue**(18); //  **INTENÇÃO explicita**

int nrAleat = gerAleat.**next**( );

**CÓDIGO FINAL**

**private** **GeradorIntAleatorio**(int min, int max) {

**this**.min = **Math**.min(min, max);

**this**.max = **Math**.max(min, max);

}

public static **GeradorIntAleatorio entre**(int min, int max) {

**return** new **GeradorIntAleatorio**(min, max);

}

public static **GeradorIntAleatorio maiorQue**(int min) {

**return new GeradorIntAleatorio**(min, **Integer**.*MAX\_VALUE*);

}

static final int VALOR\_MINIMO = 0;

public static **GeradorIntAleatorio menorQue**(int max) {

**return**.new **GeradorIntAleatorio**(**VALOR\_MINIMO**, max);

}

**1.**

Pergunta 1

Aponte na lista abaixo apenas problemas de construtores apontados nas videoaulas:

Construtores não expressam intenção

Com construtores não é possível sincronizar o acesso de dois clientes a um dado objeto

Não se consegue voltar um objeto de subclasse com construtores

Com um construtor, volta-se sempre um objeto novo da classe, com todo custo operacional inerente

Impossível ter dois construtores com mesma assinatura

O uso de construtores pode implicar em baixo acoplamento e alta coesão

Não consigo voltar o mesmo objeto outra vez com construtores

**2.**

Pergunta 2

Seja o seguinte trecho de código, usado como exemplo nesta questão:

public boolean decripta(int ateK) {

  int j = 2;

  while (j <= ateK) {

    BigInteger message = BigInteger.valueOf(j);

    // ...

  }

  // …

}

Sabe-se que a classe Java BigInteger oferece muitos métodos para se trabalhar com números inteiros muito grandes, como os usados em algoritmos criptográficos.

Com base no código exemplo acima, que conceito orientado a objetos o 'BigInteger.valueOf(Int)' corresponde?

Método estático de fábrica

Construtor

Padrão Strategy

Classe BigInteger

**3.**

Pergunta 3

Embora os nomes de Static Factory Methods devam expressar a intenção da criação de objetos de dada classe e podem, assim, serem quaisquer nomes, usualmente percebe-se que getInstance e newInstance são alguns nomes comuns para métodos de fábrica estáticos encontrados nos códigos de aplicativos e na própria API de classes utilitárias de Java. Por exemplo, usualmente getInstance( ) sem parâmetros tem o seguinte significado:

—> Retorna uma instância da classe que é distinta a cada invocação ou, no caso de uma classe Singleton, retorna a única instância dessa classe em todas as invocações.

**Obs.**: Para entendimento desta questão, apresento o objetivo do padrão de projeto Singleton: "Assegurar que uma classe tenha apenas uma instância e fornecer um ponto global de acesso a essa instância única"!

Nessa mesma linha, o significado de newInstance( ) é equivalente ao getInstance( ) no sentido de se obter uma instância da dada classe, exceto que, com newInstance( ), se deseja garantir que cada instância retornada seja distinta de todas as instâncias anteriormente retornadas.

V

F

**4.**

Pergunta 4

Uma vantagem de Static Factory Methods é que, ao contrário de construtores (em que se faz uso do nome da classe correspondente), eles podem receber nomes em consonância com a intenção do desenvolvedor de criar instâncias da classe num dado contexto. Dessa forma, um bom nome escolhido para um Static Factory Method, mas que não descreve a intenção da criação do objeto da classe, torna o seu uso mais intuitivo, deixando o código do cliente mais fácil de ler e entender!

F

V

[**Ir para o conteúdo principal**](https://www.coursera.org/learn/desenvolvimento-agil-com-padroes-de-projeto/exam/haQvj/avaliacao-semanal/attempt#main)

Parte superior do formulário

PESQUISAR NO CURSO

Pesquisar

Parte inferior do formulário

* Português (Brasil)
* **H**

Heleno Cardoso da Silva Filho

Ocultar menu

* **Introdução e Visão Geral do Curso**
* **Base Conceitual de Padrões de Projeto**
* **Introdução a Padrões de Projeto**
* **Criação de Objetos – Parte 1**
* **Classificação dos Padrões de Projeto desta Semana de Acordo com o GoF**
* **Graded Quiz**

**Avaliação Semanal**

**1.**

Pergunta 1

Quais são os quatro conceitos de Orientação a Objetos que formam a base dos Padrões de Projeto?

Abstração de Dados – Herança – Encapsulamento – Polimorfismo

Abstração de Comportamento – Herança – Encapsulamento – Polimorfismo

Abstração de Dados – Agregação – Composição – Delegação

Abstração de Comportamento – Herança – Composição – Polimorfismo

**2.**

Pergunta 2

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

Falso

Verdadeiro

**3.**

Pergunta 3

Em Java, tanto Interface quanto Classe só podem ser associadas a comportamento.

Falso

Verdadeiro

**4.**

Pergunta 4

Suponha que a classe Carro, junto com outros métodos, possui o método consertaMotor( ), que é sobreposto (overridden) na classe Carro Diesel, mas não na classe Carro Gasolina, conforme ilustrado no diagrama de classes abaixo.

Olhe o diagrama e verifique se a linha de código "carro.consertaMotor( );" dentro do método repara(Carro) da classe Mecânico é válida ou não.

Depois escolha a alternativa correta dentre as opções abaixo!

Diagrama

Descrição gerada automaticamente

Não é válida porque "consertaMotor( )" está definida na classe CarroDiesel!

Não é válida porque "consertaMotor( )" está definida na classe Carro!

Não é válida, não importando onde "consertaMotor( )" tenha sido definida!

Sempre é válida, não importando onde "consertaMotor( )" tenha sido definida!

É válida porque "consertaMotor( )" esta definida na classe CarroDiesel!

É válida porque "consertaMotor( )" está definida na classe Carro!

**5.**

Pergunta 5

Com base no diagrama abaixo, examine o seguinte trecho de código:

// ...

Estudante estudante = new EstudanteTempoIntegral( ):

Professor professor = new Professor( );

professor.ajuda(estudante);

// ...

Em seguida, verifique se os itens I e II abaixo, que estão dentro do método "ajuda(Estudante)", são válidos ou não!

I. "estudante.printHorário( );"

II. "estudante.aplicaParaEmpréstimo( );"

Finalmente, escolha a alternativa correta dentre as opções abaixo!

Diagrama

Descrição gerada automaticamente

Apenas I é válido!

Apenas II é válido!

I e II são válidos!

I e II não são válidos!

**6.**

Pergunta 6

Às vezes, queremos empregar o polimorfismo sem que seja preciso que todas as classes envolvidas estejam em uma mesma hierarquia de herança. A construção 'interface' do Java nos permite fornecer interfaces compartilhadas (isto é, coleções de operações) nessa situação. Ao fazer isso, considerando que não existam métodos default declarados na interface, não há então nenhuma implementação herdada, uma vez que cada classe deve implementar todas as operações definidas pela Interface implementada pela classe.

Falso

Verdadeiro

**7.**

Pergunta 7

Seja o seguinte trecho de código, usado como exemplo nesta questão:

public boolean **decripta**(int ateK) {

  int j = 2;

  while (j <= ateK) {

    BigInteger message = BigInteger.valueOf(j);

    // ...

  }

// …

}

Sabe-se que "valueOf(int)" é um método estático de fábrica (Static Factory Method) da classe Java BigInteger, que trabalha com números inteiros muito grandes, como os usados em algoritmos criptográficos.

Com base no código exemplo acima e no nome do método estático de fábrica, "valueOf(int)", escolha a intenção que deu origem a esse método na API da classe Java BigInteger.valueOf(j)!

Obter o valor String correspondente da variável j

Transformar valor BigInteger em Integer

Transformar valor int em BigInteger

Transformar valor BigInteger em int

**8.**

Pergunta 8

Padrões de Projeto (*Design Patterns*) podem ser definidos como soluções já testadas para problemas que ocorrem frequentemente em projetos de *software*

Falso

Verdadeiro

**9.**

Pergunta 9

O desenvolvimento de software é uma das atividades mais difíceis que o ser humano realiza. No seu caminho, o desenvolvedor encontra muitas dificuldades e problemas, ligados usualmente ao entendimento do problema e do código correspondente.

Design Patterns surgiram na busca de soluções testadas para um conjunto desses problemas, tornando-se um mecanismo eficiente no compartilhamento de conhecimento entre os desenvolvedores.

GoF (*Gang of Four*) propõe um modo de categorização dos *Design Patterns*, definindo famílias de padrões relacionados, descritas da seguinte forma:

I. Abrange a configuração e inicialização de objetos e classes.

II. Lida com as interfaces e a implementação das classes e dos objetos.

III. Lida com as interações dinâmicas entre grupos de classes e objetos.

Essas famílias são denominadas, respectivamente:

Standard Patterns, Creational Patterns e Structural Patterns

Creational Patterns, Structural Patterns e Behavioral Patterns

Standard Patterns, Structural Patterns e Behavioral Patterns

Behavioral Patterns, Structural Patterns e Standard Patterns

Structural Patterns, Standard Patterns e Creational Patterns

**10.**

Pergunta 10

O padrão de projeto Strategy define uma família de algoritmos, encapsula cada um dos algoritmos e os torna intercambiáveis; ele permite também que o algoritmo varie independentemente dos clientes que o utilizam.

Falso

Verdadeiro

Na programação orientada a objetos, existem padrões que servem de guia para que as pessoas escrevam um código mais compreensível, flexível e fácil de manter.

Esse conjunto de 5 princípios foi introduzido por Robert C. Martin, o “Uncle Bob”, em um paper denominado “Design Principles and Design Patterns”.

Posteriormente, Michael Feathers cunhou o acrônimo SOLID:

* Single Responsibility Principle (Responsabilidade Única)
* Open/Closed Principle (Aberto e Fechado)
* Liskov Substitution Principle (Substituição de Liskov)
* Interface Segregation Principle (Segregação de Interface)
* Dependency Inversion Principle (Inversão de Dependência).

POO

**1.**

Pergunta 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? **[Na sua resposta, por limitação do presente ambiente de correção automática, coloque obrigatoriamente apenas a primeira letra em maiúscula; o restante deverá estar em minúscula!]**

**Construtor**

**2.**

Pergunta 2

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

F

Correto

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

**3.**

Pergunta 3

Uma classe abstrata pode não ter nenhum método abstrato.

Falso

Verdadeiro

**4.**

Pergunta 4

O código abaixo compilará corretamente **(indique apenas e exatamente V ou Verdadeiro ou Sim – F ou Falso ou Não; e não deve justificar). Obs. : se você escrever as iniciais da sua resposta em minúsculas (v ou verdadeiro ou sim – f ou falso ou não), o sistema não entenderá e marcará como errado, mesmo se for a resposta certa!** [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.

Correto

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

**5.**

Pergunta 5

O código abaixo compilará corretamente **[supondo que o trecho representado por "..." compila corretamente]**:

class Teste{

   public void metodo (int i) {...}

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

}

Falso

Verdadeiro

Correto

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

**6.**

Pergunta 6

O código abaixo compilará corretamente **(indique apenas e exatamente V ou Verdadeiro ou Sim – F ou Falso ou Não; e não deve justificar). Obs. : se você escrever as iniciais da sua resposta em minúsculas (v ou verdadeiro ou sim – f ou falso ou não), o sistema não entenderá e marcará como errado, mesmo se for a resposta certa!** [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.**

Pergunta 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.**

Pergunta 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.**

Pergunta 9

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

Interface

Sobrecarga

Sobreposição

Método

**10.**

Pergunta 10

O princípio de \_\_\_\_\_\_\_\_\_\_\_\_ permite que objetos que pertencem a diferentes classes respondam de forma distinta a mensagens idênticas. **[Na sua resposta, por limitação do presente ambiente de correção automática, coloque obrigatoriamente apenas a primeira letra em maiúscula; o restante deverá estar em minúscula!]**

**Polimorfismo**

**11.**

Pergunta 11

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

Protegidas

Públicas

Privadas

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.

**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.

Pergunta 1

Quais são as peças ou entidades que usamos em um software orientado a objetos? [Responsa nesse formato "xxxx e yyyy", sem aspas!]

**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.

Pergunta 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.

Pergunta 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.

Pergunta 4

Objetos são criados a partir de classes

Verdadeiro

Falso

5.

Pergunta 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.

Pergunta 6

Objetos representam instâncias individuais de uma classe

Verdadeiro

Falso

7.

Pergunta 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.

Pergunta 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.

Pergunta 1

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

Função

Objeto

Classe

Herança

2.

Pergunta 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.

Pergunta 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.

Pergunta 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.

Pergunta 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

**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**

**gol**.acelerar( );

**gol**.frear( );

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

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

**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.

**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.

Caracterizar e Identificar responsabilidades de uma classe

Serve para facilitar a identificação de atributos e comportamentos de classes.

**Antropomorfismo**

É a capacidade de aplicar conceitos e características de um ser humano em objetos e animais irracionais, como por exemplo, um feijão, que possui chapéu, bolsa, toalha, possui braços, pernas, etc

Resumindo, aplicar a objeto e classes os conceitos e comportamentos próprios do homem, ou seja, do ser humano.

Exemplo, classe Homem, **ser racional**. Quais são as ações que um homem sabe ou faz?

**O homem sabe**? Nome, data de nascimento, endereço, idade, etc

**O homem faz**? Fala; anda, pula, dança, pensa, etc

Exemplo, classe Cachorro, **ser irracional**. Quais são as ações que um cachorro sabe ou faz?

**O cachorro sabe**? Local da casinha, nome, quem é seu dono, etc

**O cachorro faz**? Late; anda, pula, dança, etc

RESPONSABILIDADE

Significa o que uma classe sabe ou faz.

O que um objeto sabe = conhecimento que ele mantém = atributos, estado, variáveis de instância.

O que um objeto faz = conhecimento que ele realiza = ações, comportamento, métodos.

Quais são os dois tipos de responsabilidade existentes? Tipo ... e tipo ...!

Tipo faz e tipo sabe

Tipo sabe e tipo faz

Quais as responsabilidades de um objeto, por exemplo, uma coisa inanimada, um ponto gráfico? Conceitos típico de um ser humano a um objeto gráfico.

**O ponto gráfico sabe**? cor, posicao, tamanho, etc

**O ponto gráfico faz**? Move-se compara-se a outro ponto, devolve coordenada, etc

Através das **responsabilidades**, é possível determinar o estado = atributos, suas variáveis de instância e os comportamentos = métodos de uma determinada Classe.

Mas, um exemplo, classe Carro:

**O Carro sabe**? velocidade, potencia do motor, cor, marca, etc

**O Carro gráfico faz**? Acelerar, frear, etc

O método "Acelerar" constitui uma realização de uma responsabilidade do tipo:

Faz

Sabe

Responsabilidades:

**Tipo sabe**: estado, atributos: variáveis de instância.

**Tipo faz**: comportamentos, métodos.

**Lógica das Responsabilidades**

Definir a lógica das responsabilidades de uma classe.

Neste caso, é definir e identificar a lógica das responsabilidades de uma classe.

Dada a classe **ContaCorrente**, responsabilidades desta Classe:

**Tipo sabe**: número da conta, valor do saldo atual, etc.

**Tipo faz**: credita valor ao saldo atual, debita valor do saldo atual, é possível retirar valor? Saque ...métodos ...

**Lógica da responsabilidade**: Ações (receita), algoritmos.

**Qual a lógica da responsabilidade**, **sabe o número da conta?** Objeto **ContaCorrente** tem atributo número da conta, objeto devolve o valor do número da conta.

**É possível retirar determinado valor**? É uma **responsabilidade do tipo faz**. Responde a um método booleano.

**se** (saldo atual > valor) **então** sim **senão** não.

**Debita valor do saldo atual?** Atributo saldo atual; faz; saldo = saldo - valor

A lógica de responsabilidade do tipo faz corresponde a:

Algoritmo da responsabilidade

Receita do que deve ser realizado pela responsabilidade

Ações que a responsabilidade deve realizar

**Classe Banco:**

**Tipo sabe**: código do banco, status da conta, etc

**Sabe as** contas ativas, atributo que representa a lista de objetos de contas ativas.

**Sabe as** contas inativas, atributo que representa a lista de objetos de contas inativas.

**Tipo faz**: Registra uma conta nova, apresenta os números das contas ativas, obtém o saldo total do banco, etc.

**Responsabilidade** – Registra uma conta nova: cria um objeto conta corrente novo; insere esse objeto na lista de objetos de contas ativas.

**Qual é a lógica da responsabilidade** – **Apresenta os números das contas ativas?** Atributo das contas ativas; Algoritmo que retorna a lista de contas ativas.

**Qual é a lógica da responsabilidade** – **Obtém o saldo total do banco?** Atributo do saldo total; Algoritmo que retorna o saldo total do banco.

**Identificação de Colaborações**

Caracterizar e identificar classes colaboradoras e colaborações de uma classe.

Neste caso, é identificar as classes colaboradoras e colaborações das responsabilidades de uma classe.

Classe **ContaCorrente** – responsabilidades

**Sabe**: número da conta corrente; valor do saldo atual, status da conta, etc

**Faz**: Credita valor ao saldo atual; debita valor do saldo atual

Classe **Banco** – responsabilidades

**Sabe**: lista de contas ativas; lista de contas inativas

**Faz**: registra uma conta nova; **apresenta os números das contas ativas**; obtém o saldo total do banco.

**Lógica da Responsabilidade:**

**Apresenta os números das contas ativas?** Ação da classe Banco

Classe **Colaboradora**: ContaCorrente; **Colaboração**: Número da conta;

**Obtém o saldo total do banco?** Ação da classe Banco

Classe **Colaboradora**: ContaCorrente; **Colaboração**: valor do saldo atual;

Resumindo, para realizar a lógica de uma dada responsabilidade:

**Um objeto da classe pode realizar a tarefa sozinha** OU

**Solicitar a colaboração de objetos de classe colaboradora**.

A classe "Conta Corrente" é classe colaboradora da classe Banco porque a responsabilidade de Banco "Obtém o saldo total do Banco", para realizar o que tem de realizar, precisa da colaboração da responsabilidade "Sabe o saldo atual" da classe "Conta Corrente"!

Verdadeiro

Falso

**CARTÃO CRC – Class Responsability Card**

|  |  |
| --- | --- |
| Classe: **ContaCorrente** | |
| **Responsabilidade** | **Colaboração** |
| Credita valor do saldo atual |  |
| Debita valor do saldo atual |  |

|  |  |
| --- | --- |
| Classe: **Banco** | |
| **Responsabilidade** | **Colaboração** |
| Registra uma conta nova | Colaboradora: ContaCorrente  Colaboração: Construtor |
| Apresenta os nrs das contas ativas | Colaboradora: ContaCorrente  Colaboração: Sabe o número da conta |
| Obtém o saldo total do Banco | Colaboradora: ContaCorrente  Colaboração: Sabe o valor do saldo atual |

**Nota**: A **colaboração** corresponde a uma responsabilidade de Classe colaboradora.

A classe **colaboradora** é a classe servidora, ela vai fornecer a sua responsabilidade como um serviço para outras classes. Já o **serviço** é a **colaboração**.

Já a **classe** que **depende do serviço** é chamada de **classe cliente**.

**Representação UML – dependência/acoplamento**

Classe Servidora

**depende**

Classe Cliente

**Acoplamento - associação**

A classe cliente ela tem uma responsabilidade que depende, ela precisa da colaboração de uma responsabilidade da classe servidora/colaboradora, a relação de dependência entre a classe cliente e a classe servidora, esse conceito é chamado de acoplamento. A classe cliente está acoplada a classe servidora. Devemos tentar diminuir estes acoplamentos.

O que é colaboração, então?

Responsabilidade de classe colaboradora.

**Cartão CRC – Classe -> Responsabilidade -> Colaboração**

Conceitos de responsabilidades de uma classe, incluindo lógica de responsabilidade e de classe colaboradora e colaboração.

|  |  |
| --- | --- |
| Classe: **XXXX** | |
| **Responsabilidade** | **Colaboração** |
|  |  |

* Técnica originalmente proposta para ensinar OO -> efetiva para profissionais com pouca experiência
* Modelo CRC não pertence a UML, mas tem sido bastante utilizado com XP
* A técnica inclui uma sessão CRC que serve para extração (também modelagem de requisitos) junto a especialistas de domínio e desenvolvedores.
* Uma sessão CRC envolve por volta de meia dúzia de pessoas: especialistas de domínio, projetistas, analistas e um moderador.
* A cada pessoa é entregue um cartão CRC.

No contexto de desenvolvimento de software orientado a objetos, qual o significado da sigla CRC?

Classe Responsabilidades Colaborações

Commit Reconcilia Consiste

Componente Responsabilidade Colaborador

Classe Requisitos Conexões

Ajuda a explorar de maneira informal classes e objetos. Fornece uma introdução “mais fácil” a orientação a objetos.

De uma forma geral, auxilia a estruturar conceitos de orientação de uma maneira mais informal.

Foram criados para a aprendizagem efetiva dos conceitos de orientação a objetos. Muito utilizado na indústria.

Ponto inicial muitas metodologias de orientação de a objetos, pois facilita a identificação e caracterização das classes que serão utilizadas na aplicação. Projeto simplificado nas metodologias ágeis/modelagem ágil.

**AVALIAÇÃO**

Pergunta 1

Na prática, uma responsabilidade é alguma coisa que um objeto sabe (conhece) ou faz, sozinho ou não

Verdadeiro

Falso

**2.**

Pergunta 2

Como exemplo de antropomorfismo: Um objeto Pedido conhece sua data de realização e sabe fazer o cálculo do seu total

Falso

Verdadeiro

**3.**

Pergunta 3

Se um objeto tem uma responsabilidade a qual não pode cumprir sozinho, ele deve solicitar \_\_\_\_\_\_\_\_\_\_\_\_ de outros objetos.

colaboração

**4.**

Pergunta 4

Quando um objeto A envia uma mensagem a outro objeto B, diz o seguinte no contexto de CRC:

A é uma classe colaboradora ou servidora de B

A colabora com B

B se comunica com A

B colabora com A

**5.**

Pergunta 5

A sigla CRC significa:

Classe–Responsabilidade–Colaborações

Classe–Requisitos–Conexões

Componente–Responsabilidade–Colaborador

Commit–Reconcile–Consistency

**6.**

Pergunta 6

Responsabilidade constitui o que a classe \_\_\_\_\_\_\_\_\_\_\_ ou \_\_\_\_\_\_\_\_\_\_.

sabe/faz

**7.**

Pergunta 7

Podemos associar "responsabilidades do tipo faz" com \_\_\_\_\_\_\_\_\_\_\_\_ de uma classe!

comportamento

**8.**

Pergunta 8

Podemos associar "responsabilidades do tipo sabe" com \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ de uma classe!

estado

**9.**

Pergunta 9

Existem dois tipos de responsabilidades: responsabilidade do tipo \_\_\_\_\_\_\_\_\_ e responsabilidade do tipo \_\_\_\_\_\_\_\_\_\_\_!

sabe/faz

**10.**

Pergunta 10

A lógica de "responsabilidade do tipo faz" corresponde a:

Descrever a finalidade da classe

Receita do que deve ser feito pela responsabilidade

Algoritmo da responsabilidade ou método correspondente

Ações que a responsabilidade deve realizar

**11.**

Pergunta 11

O método "acelerar" da classe Carro constitui uma realização de uma responsabilidade do tipo \_\_\_\_\_\_\_\_\_\_\_\_\_\_!

faz

**12.**

Pergunta 12

O que é colaboração?

Classe servidora

Responsabilidade de classe colaboradora

Classe colaboradora

Responsabilidade de classe cliente

**13.**

Pergunta 13

Num projeto orientado a objetos, você identifica classes, determina as responsabilidades das classes e descreve os relacionamentos entre classes. No contexto de cartões CRC, relacionamentos entre classes corresponde a \_\_\_\_\_\_\_\_\_\_\_\_\_ entre classes!

colaboração

Qual seria a forma correta de se modelar um sistema orientado a objetos?

Poucas classes que concentram muita funcionalidade

Muitas classes onde a funcionalidade está distribuída

**Modelagem de CRC**

Modelar aplicações orientadas a objetos com o auxílio dos cartões CRC. Apresentar a modelagem de um desenvolvimento de uma aplicação, descobrir e identificar as classes, suas responsabilidades, e como elas estão relacionadas, suas colaborações.

Uma abordagem informal para modelar uma aplicação com classes e objetos. Com base na especificação dos requisitos da aplicação.

**Essa técnica consta de 6 passos:**

1. Identificar Classes e Objetos – procurar por substantivos / nomes na especificação de requisitos do sistema, serão potenciais classes e objetos.
2. Refina a lista de classes. Retirar nomes que representam atributos
3. Identificar as **responsabilidades ÓBVIAS** de uma aplicação no contexto da modelagem CRC
4. Identificar os verbos como responsabilidades na especificação de requisitos
5. Atribuição das novas responsabilidades
6. Descrever a lógica para cada responsabilidade do **tipo faz** **e** identificar as colaborações

Passo ZERO: Especifica a aplicação

**Sistema de Automação da Biblioteca - SAB**

**- Passo 1 – Identificar substantivos, nomes na especificação de requisitos do sistema, potenciais CLASSES.** Nomes representam potenciais objetos e classes.

A biblioteca tem nome, mantém um catálogo de livros, onde cada livro tem título, autor e número único de catálogo. Há usuários da biblioteca registrados, cada um com nome único de usuário. Um usuário da biblioteca pode emprestar um livro e devolver o livro emprestado para o sistema. No entanto, cada transação de empréstimo ou devolução de livro deve ser registrada por uma bibliotecária. A bibliotecária tem que registrar um novo usuário da biblioteca no SAB; quando solicitada, ela pode exibir, em ordem crescente do número único de catálogo, os livros disponíveis para empréstimo, bem como exibir os livros já emprestados e, portanto, indisponíveis para empréstimo.

Pode-se identificar potenciais objetos e classes por meio dos nomes (substantivos) ou expressões substantivas encontrados na especificação de requisitos do sistema!

Falso

Verdadeiro

Os nomes (substantivos) ou expressões substantivas encontrados na especificação de requisitos do sistema são considerados potenciais objetos e classes, os quais serão refinados posteriormente para se chegar nas classes iniciais da aplicação de software.

**- Passo 2 – Retirar nomes que representam atributos, potenciais CLASSES**

**Nota**: Identificar nomes diferentes que representam a mesma classe. Descrever o significado de cada classe. Retirar nomes que representam atributos, identificar relacionamentos entre os nomes; identificar nomes diferentes que representam mesma classe; Nomes fora do escopo do sistema e/ou são verbos enrustidos.

Ficamos então, com: **deve ser criada no singular** o **nome da CLASSE**: biblioteca, livro, usuario.

Nota: Poderia ter a classe **Bibliotecaria**, mas neste vamos considerar que não é necessário ter esta classe, uma vez que, estamos considerando como ATOR, usuário do sistema.

Exemplo de nomes fora do escopo: bibliotecária(**ator**) e ordem. Verbos enrustidos, não são substantivos: transação(**realizar uma transação**); empréstimo(**emprestar**), devolução(**devolver**).

CLASSES identificadas: **Biblioteca**, **Livro**, **Usuario**.

|  |  |
| --- | --- |
| Classe: **Biblioteca** | |
| **Descrição:** Representa o sistema de automação da biblioteca - SAB | |
| **Responsabilidade** | **Colaboração** |
|  |  |

|  |  |
| --- | --- |
| Classe: **Livro** | |
| **Descrição:** Representa livros a serem emprestados a usuários da biblioteca | |
| **Responsabilidade** | **Colaboração** |
|  |  |

|  |  |
| --- | --- |
| Classe: **Usuario** | |
| **Descrição:** Representa usuários que emprestam livros da biblioteca | |
| **Responsabilidade** | **Colaboração** |

**- Passo 3 – Identificar as responsabilidades e colaborações, CLASSES**

Identificar as responsabilidades óbvias para a classe.

|  |  |
| --- | --- |
| Classe: **Biblioteca** | |
| **Descrição:** Representa o sistema de automação da biblioteca - SAB | |
| **Responsabilidade** | **Colaboração** |
| **Sabe**: nome, catálogo de livros, lista de usuários  **Faz**: Registra usuário; adiciona livro ao catálogo |  |

|  |  |
| --- | --- |
| Classe: **Livro** | |
| **Descrição:** Representa livros a serem emprestados a usuários da biblioteca | |
| **Responsabilidade** | **Colaboração** |
| **Sabe**: título, autor, nr. único do catálogo de livro  disponibilidade de empréstimo do livro,  usuário, se emprestado |  |

|  |  |
| --- | --- |
| Classe: **Usuario** | |
| **Descrição:** Representa usuários que emprestam livros da biblioteca | |
| **Responsabilidade** | **Colaboração** |
| **Sabe**: nome  **Faz**: lista de livros emprestados ao usuário |  |

**- Passo 4 – Identificar os verbos como responsabilidades na especificação de requisitos – Tipo faz, independentemente de qual classe ela pertence.**

Quais os verbos de ação que se encontram na especificação de requisitos.

A biblioteca tem nome, mantém um catálogo de livros, onde cada livro tem título, autor e número único de catálogo. Há usuários da biblioteca registrados, cada um com nome único de usuário. Um usuário da biblioteca pode emprestar um livro e devolver o livro emprestado para o sistema. No entanto, cada transação de empréstimo ou devolução de livro deve ser registrada por uma bibliotecária. A bibliotecária tem que registrar um novo usuário da biblioteca no SAB; quando solicitada, ela pode exibir, em ordem crescente do número único de catálogo, os livros disponíveis para empréstimo, bem como exibir os livros já emprestados e, portanto, indisponíveis para empréstimo.

**Listar os verbos da especificação – responsabilidades do tipo faz**

* Emprestar livro
* Devolver livro
* Registrar novo usuário
* Exibir livros disponíveis para empréstimo
* Exibir livros indisponíveis para empréstimo

Podemos identificar potenciais responsabilidades de classes examinando os nomes (substantivos) ou expressões substantivas encontrados na especificação de requisitos do sistema!

Falso

Verdadeiro

Não confundir, identificar **potenciais responsabilidades de classes** com **identificar potenciais classes**, em que se examinam os **nomes (substantivos)** ou **expressões substantivas** encontrados na especificação de requisitos do sistema!

O erro mais comum é inverter o que indicam os **verbos de ação** – **potenciais responsabilidades** – com o que indicam os nomes (substantivos) ou expressões substantivas – **potenciais classes** –, os dois tipos encontrados na **especificação de requisitos do sistema**!

Podemos identificar **potenciais responsabilidades** de classes examinando os **verbos de ação** encontrados na especificação de requisitos do sistema!

**- Passo 5 – Atribuição de novas responsabilidades**

Para cada **potencial responsabilidade**, verifique qual classe deve ser atribuída:

* Se corresponder a responsabilidade óbvia, buscar outra classe, por causa do polimorfismo
* Se não, atribuir a essa classe.

Portanto, os potenciais responsabilidades identificas na **fase 4**, devem fazer parte das ações da classe Biblioteca.

* Emprestar livro
* Devolver livro
* **Registrar novo usuário**, **já existe na CLASSE**
* Exibir livros disponíveis para empréstimo
* Exibir livros indisponíveis para empréstimo

|  |  |
| --- | --- |
| Classe: **Biblioteca** | |
| **Descrição:** Representa o sistema de automação da biblioteca - SAB | |
| **Responsabilidade** | **Colaboração** |
| **Sabe**: **nome, catálogo de livros, lista de usuários**  **Faz**: **Registra novo usuário; adiciona novo livro ao catálogo**.  **Emprestar livro, devolver livro, exibir livros disponíveis para empréstimo, exibir livros indisponíveis para empréstimo** |  |

**- Passo 6 – Descrever a lógica de cada responsabilidade do tipo faz e identificar as colaborações**

- Quais são as ações necessárias a serem realizadas na responsabilidade?

- **Lógica – Emprestar livro de biblioteca** - **algoritmo**

* Tem o livro para ser emprestado para o usuário; **sabe o livro e o usuário**
* ~~Marcar livro como emprestado~~

**Novas responsabilidades**

* Anexar usuário como emprestador do livro
* Anexar livro à lista de livros de usuários

- **Lógica – Devolver livro a biblioteca** - **algoritmo**

* Desanexar livro da lista de livros emprestados pelo usuário – **classe Usuario**
* Desanexar usuário como alvo do empréstimo pelo usuário **– classe Livro**
* ~~Marcar livro como disponível~~ **– classe Livro**

**Novas responsabilidades**

- **Lógica – Exibir livros disponíveis** - **algoritmo**

* Para cada livro do catálogo de livros

**Novas responsabilidades**

**se** (**livro está disponível?**) // **classe Livro**

**então** coloca na lista para exibir. // **sabe disponibilidade de empréstimo**.

- **Lógica – Exibir livros não disponíveis** - **algoritmo**

* Para cada livro do catálogo de livros

**mesma respons. anterior**

**se** (**livro não está disponível?**) // **classe Livro**

**então** coloca na lista para exibir. = **sabe disponibilidade de empréstimo**.

- **Lógica – Registra usuário** - **algoritmo**

* **se** (**usuário não está na lista de usuários da biblioteca?**) // **classe Livro**

**então** cria/instancia um novo objeto Usuario.

adiciona esse objeto à lista de usuários da biblioteca e avisa “novo usuário”

**caso contrário** avisa “usuário já existente”

**Lógica – Adiciona livro** - **algoritmo**

* cria/instancia um objeto Livro sob o número único de catálogo

adiciona objeto ao catálogo de livros da biblioteca

|  |  |
| --- | --- |
| Classe: **Biblioteca** | |
| **Descrição:** Representa o sistema de automação da biblioteca - SAB | |
| **Responsabilidade** | **Colaboração** |
| **Sabe**: **nome, catálogo de livros, lista de usuários**  **Faz**: **Registra novo usuário; adiciona novo livro ao catálogo**.  **Emprestar livro, devolver livro, exibir livros disponíveis para empréstimo, exibir livros indisponíveis para empréstimo** |  |

|  |  |
| --- | --- |
| Classe: **Livro** | |
| **Descrição:** Representa livros a serem emprestados a usuários da biblioteca | |
| **Responsabilidade** | **Colaboração** |
| **Sabe**: **título, autor, nr. único do catálogo - livro**  **disponibilidade de empréstimo do livro**  **usuário, se emprestado**  **~~Marcar livro como emprestado~~\*\*\***  **Anexar usuário como alvo do empréstimo do livro**  **Desanexar usuário como alvo do empréstimo**  **~~Marcar livro como disponível~~ \*\*\*** |  |

|  |  |
| --- | --- |
| Classe: **Usuario** | |
| **Descrição:** Representa usuários que emprestam livros da biblioteca | |
| **Responsabilidade** | **Colaboração** |
| **Sabe**: **nome**  **Faz**: **lista de livros emprestados ao usuário**  **Anexa livro à lista de livros emprestados**  **Desanexa livro da lista de livros emprestados** |  |

**Identificar as colaborações**

**Novas responsabilidades,** apareceram no momento da definição da lógica da responsabilidade

Não foram identificas nas responsabilidades óbvias e nem quando foi revisado as novas responsabilidades com base no requisito **no passo 4**. Só apareceram na descrição da lógica.

Identificar quais as classes colaboradoras devem ser atribuídas as responsabilidades?

**Novas responsabilidades - Acoplamento**

**CLASSE Livro**

* Marcar livro como emprestado
* Anexar usuário como emprestador do livro

**CLASSE Usuario**

* Anexar livro à lista de livros de usuários

**Resumo Colaboração**

|  |  |
| --- | --- |
| Classe: **Biblioteca** | |
| **Descrição:** Representa o sistema de automação da biblioteca - SAB | |
| **Responsabilidade** | **Colaboração** |
| Registra usuário | **Colaboradora**: Usuario  **Colaboração**: Construtor |
| Adiciona livro ao catálogo | **Colaboradora**: Livro  **Colaboração**: Construtor |
| Empresta Livro | **Colaboradora**: Livro e Usuario  **Colaboração**: anexa usuario, anexa livro |
| Devolve livro | **Colaboradora**: Livro e Usuario  **Colaboração**: desanexa livro, desanexa usuario |
| Exibe livros disponíveis empréstimo | **Colaboradora**: Livro  **Colaboração**: sabe disponibilidade empréstimo |
| Exibe livros indisponíveis empréstimo | **Colaboradora**: Livro  **Colaboração**: sabe disponibilidade empréstimo |

|  |  |
| --- | --- |
| Classe: **Livro** | |
| **Descrição:** Representa livros a serem emprestados a usuários da biblioteca | |
| **Responsabilidade** | **Colaboração** |
| ~~Marca livro como emprestado~~ | **Colaboradora**:  **Colaboração**: |
| Anexa usuário do empréstimo | **Colaboradora**:  **Colaboração**: |
| Desanexa usuário do empréstimo | **Colaboradora**:  **Colaboração**: |
| ~~Marca livro como disponível~~ | **Colaboradora**:  **Colaboração**: |

|  |  |
| --- | --- |
| Classe: **Usuario** | |
| **Descrição:** Representa usuários que emprestam livros da biblioteca | |
| **Responsabilidade** | **Colaboração** |
| Anexa livro à lista de livros emprestados | **Colaboradora**:  **Colaboração**: |
| Desanexa livro da lista de livros emprestados | **Colaboradora**:  **Colaboração**: |

**Diagrama de colaboração**

**depende**

Classe **Livro**

Classe **Biblioteca**

**depende**

Classe **Usuario**

Pergunta 1

Pode-se identificar potenciais objetos e classes por meio dos seguintes elementos gramaticais encontrados na especificação de requisitos do sistema:

expressão substantiva

nome (substantivo)

verbo

Adjetivo

Pergunta 2

O que é Modelagem CRC?

Uma abordagem sistemática para modelar as classes e objetos de uma aplicação sem considerar o uso de cartões CRC

Uma abordagem formal para modelar as classes e objetos de uma aplicação com base em cartões CRC

Uma abordagem informal para modelar as classes e objetos de uma aplicação com base em cartões CRC

Pergunta 3

Por convenção, o nome de uma classe não pode ser plural

Verdadeiro

Falso

Pergunta 4

Para se identificar as responsabilidades óbvias de uma classe, parte-se da \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dessa classe!

funcionalidade

Pergunta 5

Pode-se identificar potenciais responsabilidades de duas maneiras:

Dada a especificação de requisitos do sistema, identificam-se verbos que indicam ação, acontecimento, fenômeno natural, desejo e atividade mental.

Dada uma classe, procura-se identificar responsabilidades óbvias

Dada a especificação de requisitos do sistema, identificam-se diversos elementos gramaticais, tais como substantivos, adjetivos e verbos

Pergunta 6

Imagine um sistema bancário com duas classes: Banco e Conta; imagine que a classe Conta tenha as seguintes responsabilidades: "credita conta", "debita conta" e "obtém saldo corrente"; imagine que a classe Banco tenha a seguinte responsabilidade: "fornece saldo total". Ao definir a lógica da responsabilidade "fornece saldo total", você percebe que a responsabilidade "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" da classe Conta é sua colaboração!

credita conta

A classe Conta nem é classe colaboradora da classe Banco

debita conta

obtém saldo corrente

**Compilando em JAVA – java – para executar; javac – para compilar**

**Nota**: Precisa ter o SDK + JVM + IDE

public class **HelloWorld**{

public **static void main**(String[] args){

System.out.**println**(“Hello World!”);

}

}

**Nota:** Salvar o arquivo igual ao nome da classe HelloWorld.java

**Compilando**: **javac** HelloWorld.java

Que comando eu utilizo para compilar um programa em Java?

javac

**Executando**: **java** HelloWorld

Que comando eu utilizo para executar uma classe em Java?

java

**Métodos e Atributos Estáticos**

Criar e utilizar métodos e atributos estáticos em classes, sabendo diferenciá-los dos métodos e atributos de instância.

Informações que diz respeito a todos os objetos de uma classe ou informações que diz respeito a um objeto específico.

Algumas **classes** possuem **variáveis que compartilhadas** com todas as **instâncias**.

**Instâncias** possuem valores que são diferentes para cada uma delas.

**Exemplo**. Classe Gato, o atributo número total de gatos. É um estado/informação, que é da classe Gato, mas não diz respeito a um gato específico, diz respeito a todos os gatos.

Já o atributo cor de um gato, diz respeito a um gato específico e não a todos os gatos.

A variável precisa ter o modificador “**static**”.

É uma variável da classe e não uma variável do objeto. **Exemplo**:

public **class** Gato {

**static** int totalGatos = 0; // **o mesmo valor é compartilhado por qualquer instância**

Gato( ) { totalGatos++;}

}

**Gato.totalGatos**, **não precisa** **de uma instância** para **acessar um membro estático**.

Considere uma classe Funcionario. Quais informações, se armazenadas nessa classe, poderiam ser variáveis estáticas?

Nome do funcionário

Salário

Quantidade de funcionários

A informação é a mesma para qualquer funcionário

Quem seria o funcionário do Mês

É o mesmo independente de quem é o funcionário

Seção em que trabalha

Quem é o seu supervisor

Limite salarial

**Nota**: Informação que é a mesma para qualquer funcionário.

**Atenção**: Não utilize variáveis estáticas como variáveis globais. Tem um comportamento de variável GLOBAL.

Utilizar como variáveis dentro de um escopo pertinente a uma CLASSE, mas, não como uma variável GLOBAL dentro de um contexto orientado a objetos.

Os métodos estáticos são similares as funções na programação estruturada.

public class Calc {

static int **quad**r(int i) {

return i \* i;

}

}

acelerar(carro ); // **pensamento estruturado**

carro.acelerar( ); // **pensamento orientado a objetos**

**Nota**: Sempre desconfie quando passar um objeto como parâmetro para um método estático, se de repente aquele método deve estar no objeto.

**Dica**: Se você utilizar o “**static import**”, você pode utilizar os métodos estáticos sem referenciar o nome da classe.

import static java.lang.Math.\*;

Math.**max**(a,b); ou **max**(a,b); // **Se importar o java.lang.Math**

Exemplo:

**public** **class** Somador {

String nome;

**int** valorInstancia = 0;

**static** **int** *valorEstatica* = 0;

**void** somar(){

valorInstancia++;

*valorEstatica*++;

}

**void** imprimir(){

System.***out***.println("O somador "+nome+": instância= "+valorInstancia+" e estática="+*valorEstatica*);

}

}

**public** **class** Principal {

**public** **static** **void** main(String[] args) {

Somador s1 = **new** Somador();

Somador s2 = **new** Somador();

Somador s3 = **new** Somador();

s1.nome = "Somador 1";

s2.nome = "Somador 2";

s3.nome = "Somador 3";

**for** (**int** i = 0; i<3; i++) {

s1.somar();

s2.somar();

s3.somar();

}

**//** s2.somar();

s1.imprimir();

s2.imprimir();

s3.imprimir();

}

}

Qual você acha que será o valor da variável em cada das instâncias? 3

Cada instância será incrementada de forma separada

Qual será o valor impresso para a variável estática? 9

A variável estática será incrementada independente da instância

Na programação estruturada o desenvolvedor pensa em passar tudo como parâmetro, enquanto na orientada a objetos, pensa em invocar o método no objeto.

**Programação Estruturada**

Parâmetros => funções. Pode receber uma lista como parâmetro e a função retornar a lista ordenada.

Variável global, a função também pode fazer uso.

Programação Orientada a Objetos

Classe => criamos Objetos, podemos utilizar construtores para passar parâmetros e parametrizar os objetos. Cada objeto tem sua própria informação e cada um se comporta de uma forma.

Um objeto executa seus métodos com base nos parâmetros recebidos e em seus dados internos.

O objeto pega o estado interno para realizar seu processamento.

Classe Carinho => objeto carrinho

**Valor total**? Objeto carrinho, vai responder com base nas informações internas do objeto, lista de produtos que ele possui.

**Valor do frete**? Não sabe responder, não tem informação do endereço de entrega, pois não faz parte do objeto carrinho e sim de outro objeto, faz parte do objeto Compra, classe Endereço.

Depende do peso e volume dos produtos que faz parte do escopo do objeto, mas também do endereço de entrega que não faz parte.

Informações fora da classe podem ser passadas como parâmetro, neste caso, passar o endereço como parâmetro.

Valor do frete? **Classes: Compra; Endereco; Carinho**

Para a classe Compra, sem parâmetro, para calcular o valor do frete.

O método valor do frete pode ser parte de outras classes com diferentes escopos.

Faz sentido termos métodos similares em classes diferentes?

Sim

Não

Faz sentido, quando o contexto é diferente. Uma classe Aluno pode ter um método para calcular a nota. Uma classe Turma pode ter um método de mesmo nome, mas para um contexto diferente. Nesse caso, o método de Turma chamaria o método de cada aluno.

Avaliar que dados fazem parte do escopo da classe e quais serão recebidos como parâmetros pelos métodos.

**Exemplo POO Estruturada**

**public** **class** Aluno {

**int** bim1;

**int** bim2;

**int** bim3;

**int** bim4;

}

**public** **class** VerificadorNotas {

**public** **static** **int** mediaAluno(Aluno a) {

**int** total = 0;

total += a.bim1;

total += a.bim2;

total += a.bim3;

total += a.bim4;

**return** total / 4;

}

**public** **static** **boolean** passouDeAno(Aluno a) {

**int** media = *mediaAluno*(a);

**if** (media >= 6)

**return** **true**;

**return** **false**;

}

}

**public** **class** Principal {

**public** **static** **void** main(String[] args) {

Aluno julia = **new** Aluno();

julia.bim1 = 70;

julia.bim2 = 60;

julia.bim3 = 90;

julia.bim4 = 80;

System.***out***.println(VerificadorNotas.*mediaAluno*(julia));

System.***out***.println(VerificadorNotas.*passouDeAno*(julia));

}

}

**Exemplo POO Orientada a Objetos**

**public** **class** Aluno {

**int** bim1;

**int** bim2;

**int** bim3;

**int** bim4;

**public** **int** media() {

**int** total = 0;

total += bim1;

total += bim2;

total += bim3;

total += bim4;

**return** total / 4;

}

**public** **boolean** passouDeAno() {

**if** (*media*() >= 6)

**return** **true**;

**return** **false**;

}

}

**public** **class** Principal {

**public** **static** **void** main(String[] args) {

Aluno julia = **new** Aluno();

julia.bim1 = 70;

julia.bim2 = 60;

julia.bim3 = 90;

julia.bim4 = 80;

System.***out***.println(julia.*media*());

System.***out***.println(julia.*passouDeAno*());

}

}

**Nota**: O dev estruturado está acostumado a criar A classe mais como uma estrutura de dados e utilizar uma série de classe de funções que utiliza da estrutura de dados.

**1.**

Pergunta 1

Qual o **operador** que utilizamos para instanciar o objeto de uma classe?

create

add

instance

new

**2.**

Pergunta 2

Pode haver classes com o mesmo nome?

Nunca

Sim, se estiverem em pacotes diferentes

Sim, se possuírem construtores diferentes

Sim, mas apenas em programas diferentes

Sim, mas não pode ser o mesmo nome de uma das classes da biblioteca básica do Java

**3.**

Pergunta 3

Uma classe sempre tem um construtor?

Sim

Não

Se nenhum construtor for explicitamente definido pelo programador da classe, um construtor padrão, que não recebe argumentos, é incluído para a classe pelo compilador Java.

**4.**

Pergunta 4

Quando uma classe tem mais de um construtor, o que não deve ser igual entre eles?

Retorno

Nome

Tipos dos parâmetros

Quantidade de parâmetros

**5.**

Pergunta 5

Como deve ser o papel de uma classe em um software orientado a objetos?

Ela deve implementar parte da funcionalidade do sistema e colaborar com outras classes para completar a funcionalidade delas

Ela deve possuir vários métodos que implementam diferentes partes do sistema e centralizar a invocação deles

**6.**

Pergunta 6

Marque a alternativa correta sobre um atributo estático:

Deve ser utilizado como uma variável global

Não precisa de uma instância da classe para ser acessado

A modificação feita em uma instância não afeta a outra

Fica ligada a cada instância da classe

**7.**

Pergunta 7

Marque a alternativa correta sobre métodos estáticos:

Precisa de uma instância para ser invocado

Pode acessar variáveis de instância

Pode ser utilizado o *import static* para que não precise ser referenciado o nome da classe

Precisa acessar uma variável estática

**8.**

Pergunta 8

Qual o problema no método abaixo?

public **static** double **calcularSalario**(Funcionario f){

 double semImposto = f.salarioBruto()\*(1-f.aliquotaImposto());

**return** semImposto + f.bonus();

}

Não poderia ser um método estático

Deveria ser um método de instância na classe Funcionario

Não deveria ter um retorno

Um objeto não pode ser passado como parâmetro

**EXERCÍCIO AVA – Classe para cálculo de IMC**

1. Implemente no Eclipse uma classe chamada Paciente que possui um construtor que recebe o seu peso em quilos e sua altura em metros, ambos utilizando o tipo double. Crie um método chamado calcularIMC() que calcula o índice de Massa Corporal de acordo com a fórmula IMC = peso (quilos) / (altura \* altura (metros)). Crie também um método chamado diagnostico() que utiliza o método calcularIMC() e retorna uma String de acordo com as seguintes faixas de valor:

* Baixo peso muito grave = IMC abaixo de 16 kg/m²
* Baixo peso grave = IMC entre 16 e 16,99 kg/m²
* Baixo peso = IMC entre 17 e 18,49 kg/m²
* Peso normal = IMC entre 18,50 e 24,99 kg/m²
* Sobrepeso = IMC entre 25 e 29,99 kg/m²
* Obesidade grau I = IMC entre 30 e 34,99 kg/m²
* Obesidade grau II = IMC entre 35 e 39,99 kg/m²
* Obesidade grau III (obesidade mórbida) = IMC igual ou maior que 40 kg/m²

Implemente no **STS Tool 4**, uma classe chamada Principal em que se criam 3 instâncias da classe Paciente com valores diferentes e se imprime no console o resultado dos dois métodos criados.

O índice de massa corporal, mais conhecido pela sigla IMC, é um índice adotado pela OMS (Organização Mundial de Saúde), que é usado para o diagnóstico do sobrepeso e da obesidade. O IMC pode ser facilmente calculado a partir de dois simples dados: peso e altura.

O índice de massa corporal é um relevante indicador de saúde, amparado por vários estudos, que comprovam que, em geral, quanto maior for o IMC de um indivíduo, mais elevado é o risco de morte precoce, principalmente por doenças cardiovasculares.

O IMC é um índice válido para identificar o excesso ou a carência de peso em qualquer pessoa a partir dos 2 anos de idade. Seus resultados são bastante confiáveis, mas a sua principal falha é o fato dele poder superestimar a quantidade de gordura em pessoas que tenham muito peso devido a uma grande massa muscular, como são os casos de atletas de alto rendimento e fisiculturistas.

1. Crie uma classe Pizza que possua o método adicionaIngrediente() que recebe uma String com o ingrediente a ser adicionado. Essa classe também deve possuir o método getPreco() que calcula da seguinte forma: 2 ingredientes ou menos custam 15 reais, de 3 a 5 ingredientes custam 20 reais e mais de 5 ingredientes custa 23 reais.

É preciso contabilizar os ingredientes gastos por todas as pizzas! Utilize uma variável estática na classe Pizza para guardar esse tipo de informação (dica: utilize a classe HashMap para guardar o ingrediente como chave e um Integer como valor). Crie o método estático contabilizaIngrediente() para ser chamado dentro de adicionaIngrediente() e fazer esse registro.

Crie uma nova classe chamada CarrinhoDeCompras que pode receber objetos da classe Pizza. Ela deve ter um método que retorna o valor total de todas as pizzas adicionadas. O Carrinho não pode aceitar que seja adicionada uma pizza sem ingredientes.

Crie uma classe Principal com o método main() que faz o seguinte:

* Cria 3 pizzas com ingredientes diferentes
* Adiciona essas Pizzas em um CarrinhoDeCompra
* Imprime o total do CarrinhoDeCompra
* Imprime a quantidade utilizada de cada ingrediente

**Qual o preço da pizza?**

**1.**

Pergunta 1

O que é um teste de unidade?

Que testa uma única classe ou um único método de classe do software

Qualquer teste automatizado com uma ferramenta como o JUnit é de unidade

Que unifica vários cenários em um único teste

Que testa o software completo como uma unidade

**2.**

Pergunta 2

Como é chamado o tipo de teste que verifica a funcionalidade de várias classes em conjunto?

Teste de Caixa Branca

Teste de Aceitação

Teste de Unidade

Teste de Desempenho

Teste de Integração

**3.**

Pergunta 3

Qual anotação deve ser colocada antes de um método para ele ser considerado um método de teste no JUnit?

@Test

**4.**

Pergunta 4

Como comparar em um teste que o valor retornado é o mesmo que o esperado?

Com um método da classe Assert

Com operadores como ==

Utilizando um if

Imprimindo no console

**5.**

Pergunta 5

Em uma classe de teste com 5 testes, quantas vezes um método com a anotação @BeforeClass seria executado? **[Responda um valor númerico apenas!]**

**1**

**6.**

Pergunta 6

Considere que uma classe de teste tenha apenas um teste e métodos com @Before, @After, @BeforeClass e @AfterClass. Qual seria a ordem de execução desses métodos?

@BeforeClass, @Before, @AfterClass e @After

@Before, @BeforeClass, @AfterClass e @After

@Before, @BeforeClass, @After e @AfterClass

@BeforeClass, @Before, @After e @AfterClass