UNIVERSIDADE FEDERAL DO CARIRI
TECNÓLOGO EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

2ª ATIVIDADE AVALIATIVA

Professor	Ricardo Ferreira Vilela
Disciplina:	Programação Orientada a Objetos
Semestre Letivo:	2023-1

Nesta série de exercícios, você terá a chance de aprimorar suas habilidades em Programação Orientada a Objetos, com foco nos conceitos essenciais de Herança e Polimorfismo. Os exercícios são construídos em torno de um conjunto de classes e interfaces projetadas para simular uma versão de um sistema bancário e de cartões de crédito.

Instruções Gerais:

- Certifique-se de que os nomes das classes, interfaces, atributos e funções correspondam >>>**EXATAMENTE<<<** ao texto, pois serão avaliados por meio de testes automatizados;
- Cada classe e interface solicitada deve ser criada em um arquivo separado;

Lembre-se de que o objetivo da atividade é, além da avaliação, uma prática na construção de código orientado a objetos, portanto, evite consultar soluções prontas.

EXERCÍCIOS

1) Nesta aplicação será necessária a criação de múltiplos Bancos, e para que você garanta que todos esses bancos possuam uma mesma estrutura, será necessário estabelecer um contrato para a aplicação. Em Orientação a Objetos, os contratos são estabelecidos por meio de interfaces. Portanto, o primeiro passo no desenvolvimento da aplicação será a criação das interfaces.

   A interface que estabelece o contrato para as classes bancárias deve ser chamada de *IBank*. Esta interface impõe a necessidade de que todas as classes que a implementem forneçam os métodos *deposit* e *withdraw*, ambos com visibilidade pública. O método *deposit* deve sempre retornar um valor do tipo *double* quando chamado, enquanto o método *withdraw* deve retornar um valor booleano. Ambos os métodos devem incluir um parâmetro do tipo *double*, representando o valor a ser depositado no primeiro método e o valor a ser sacado no segundo método.

2) No exercício anterior, você definiu uma interface (um conjunto de requisitos) que deveria ser cumprida por todos os bancos a serem posteriormente implementados em sua aplicação. No entanto, surgiram novos requisitos devido à necessidade de lidar com bancos que também

atuam como seguradoras. Esses bancos devem também fornecer serviços de seguro, abrangendo áreas como seguros de vida, seguros de automóveis e seguros de viagens.

Uma vez que nem todos os bancos oferecem serviços de seguros, não é viável incluir essas funcionalidades na interface *IBank* existente. Portanto, torna-se imperativo criar uma nova interface para acomodar essas características adicionais.

A interface designada para estabelecer o contrato das entidades que oferecem serviços de seguros será denominada *IInsuranceCompany*. Esta interface estipula a obrigatoriedade de que todas as classes de bancos que a implementem forneçam o método *createInsurancePolicy*.

O arquiteto da equipe, aproveitando as vantagens do polimorfismo, determinou que o método *createInsurancePolicy* será utilizado para solicitar todos os tipos de seguros. Além disso, ele estabeleceu que a distinção entre os tipos de seguro será feita com base na assinatura do método. Para isso, foram definidas as seguintes diretrizes que devem ser seguidas na interface:

- **Seguro de Vida:** Deve-se fornecer os parâmetros *name* e *age*, nessa ordem, com os tipos String e inteiro, respectivamente.
- **Seguro de Carros:** Deve-se fornecer os parâmetros *carModel*, *carID* e *manufacturedYear*, nessa ordem, com os tipos *String*, inteiro e inteiro, respectivamente.
- **Seguro Viagem:** Deve-se fornecer os parâmetros, nessa ordem, *passengerName* (String), *origin* (String), *destination* (String), *departureDate* (Date) e *returnDate* (Date).

3) Com as interfaces estabelecidas, o próximo passo no desenvolvimento consiste na implementação de um novo banco. A classe que representa esse banco deve ser nomeada como *Santander*. É importante observar que este banco atualmente não oferece serviços de seguros, o que implica que ele deve implementar apenas a interface padrão para bancos.

A classe deverá ser composta pelos seguintes atributos:

Atributo	**Tipo**	**Contexto**
name	*Cadeira de Caracteres*	Nome do Banco
manager	*Cadeira de Caracteres*	Gerente do Banco
country	*Cadeira de Caracteres*	País onde está o Banco
address	*Cadeira de Caracteres*	Endereço do Banco
balance	*double*	Patrimonio do Banco
currency	*Currency (java.util.Currency)*	Moeda padrão do Banco

O construtor da classe *Santander* deve incluir todos os atributos da classe na mesma ordem em que foram apresentados na tabela, à exceção do atributo *currency*. Para o atributo *currency*, um último parâmetro deve ser adicionado ao construtor, sendo do tipo String e denominado *currencyCode*. Diferentemente dos outros atributos, que são inicializados com os valores recebidos pelos parâmetros do construtor, a inicialização do atributo *currency* deve ser executada por meio de uma chamada à função *getInstance*() da classe *Currency*. Essa função deve ser invocada com o parâmetro *currencyCode* passado a ela. O valor retornado por essa função será o valor utilizado para a inicialização da variável *currency*.

Dica: Ao testar o funcionamento do código, certifique-se de utilizar valores válidos para a variável *currencyCode*. Você pode encontrar esses códigos ao consultar a **ISO 4217**. Por exemplo, para usar a moeda Real brasileira, insira o valor "BRL" como *currencyCode*. Isso garantirá que o código funcione corretamente com a moeda desejada.

Conforme especificado na interface, a classe *Santander* deve ser composta pelos métodos *deposit* e *withdraw*. O método *deposit* deve adicionar o valor passado como parâmetro ao saldo do banco e, em seguida, retornar o valor atualizado do saldo após o depósito. Quanto ao método de saque (*withdraw*), ele deve verificar se o valor fornecido como parâmetro é menor ou igual ao saldo atual. Se o valor solicitado para o saque for superior ao saldo, uma mensagem de erro deve ser exibida da seguinte maneira:

```
Insufficient funds!
```

Logo após a mensagem, a função deve retornar *false*.

Por outro lado, se o valor passado como parâmetro no método *withdraw* for menor ou igual ao saldo atual (balance), o valor do saldo deve ser reduzido pelo valor do saque e, em seguida, o método deve retornar a seguinte mensagem de sucesso:

```
Amount successfully withdrawn.
```

Logo após a mensagem, a função deve retornar *true*.

4) Após a criação do primeiro banco, a próxima etapa envolve a implementação de um novo banco, o qual também oferece serviços de seguros. A classe que representa esse banco deve ser nomeada como *JPMorgan*. Como este é um banco que atua como uma seguradora, a classe deve implementar todas as interfaces definidas anteriormente, ou seja, tanto a

interface *IBank* quanto a interface *IInsuranceCompany*. Isso garantirá que o *JPMorgan* cumpra os requisitos tanto de um banco quanto de uma seguradora.

A classe deverá ser composta pelos seguintes atributos:

Atributo	Tipo	Contexto
name	*Cadeira de Caracteres*	Nome do Banco
manager	*Cadeira de Caracteres*	Gerente do Banco
country	*Cadeira de Caracteres*	País onde está o Banco
address	*Cadeira de Caracteres*	Endereço do Banco
balance	*double*	Patrimonio do Banco
currency	*Currency (java.util.Currency)*	Moeda padrão do Banco

O construtor da classe *JPMorgan* deve incluir todos os atributos da classe na mesma ordem em que foram apresentados na tabela, à exceção do atributo *currency*. Para o atributo *currency*, um último parâmetro deve ser adicionado ao construtor, sendo do tipo String e denominado *currencyCode*. Diferentemente dos outros atributos, que são inicializados com os valores recebidos pelos parâmetros do construtor, a inicialização do atributo *currency* deve ser executada por meio de uma chamada à função *getInstance()* da classe *Currency*. Essa função deve ser invocada com o parâmetro *currencyCode* passado a ela. O valor retornado por essa função será o valor utilizado para a inicialização da variável *currency*.

Conforme especificado na interface, a classe *JPMorgan* deve ser composta pelos métodos *deposit* e *withdraw*. O método *deposit* deve adicionar o valor passado como parâmetro ao saldo do banco e, em seguida, retornar o valor atualizado do saldo após o depósito. Quanto ao método de saque (*withdraw*), ele deve verificar se o valor fornecido como parâmetro é menor ou igual ao saldo atual. Se o valor solicitado para o saque for superior ao saldo, uma mensagem de erro deve ser exibida da seguinte maneira:

```
Insufficient funds!
```

Logo após a mensagem, a função deve retornar *false*.

No entanto, se o valor for igual ou menor que o saldo disponível (*balance*), o saldo será reduzido pelo valor do saque e também por uma taxa de 1% sobre o valor do saque. Após a dedução, a seguinte mensagem será retornada:

```
Amount successfully withdrawn.
```

Logo após a mensagem, a função deve retornar *true*.

Após a conclusão dos métodos estabelecidos pela interface *IBank*, será necessário implementar os métodos exigidos pela interface *IInsuranceCompany*. Como mencionado anteriormente, a distinção entre esses métodos reside em suas assinaturas, mas todos eles têm o mesmo nome, que é *createInsurancePolicy*.

Na classe *JPMorgan*, é necessário implementar os métodos *createInsurancePolicy* seguindo as regras abaixo:

a) Para o **seguro de vida**, o método deve retornar a mensagem Life insurance successfully processed! e, em seguida, retornar *true*;

b) Para o **seguro de carro**, o método deve verificar o ano de fabricação do carro para determinar se ele é elegível para seguro. Apenas carros com ano de fabricação igual ou posterior a **2000** devem ser segurados. Se o carro tiver um ano de fabricação anterior a 2000, o método deve exibir a mensagem Unauthorized insurance for the car manufactured in the year [ano]. (onde [ano] é o ano do carro recebido como parâmetro) e, em seguida, retornar *true*. Se o ano do carro for igual ou superior a 2000, o método deve exibir a mensagem Car insurance successfully completed. e, em seguida, retornar *true*;

c) Para o **seguro de viagem**, o método deve retornar a mensagem Travel insurance successfully processed! e, em seguida, retornar *true*.

5) Após a conclusão da implementação dos bancos, o cliente solicitou a inclusão da seguinte funcionalidade:

> *"Como responsável pelo cadastro de cartões de crédito, gostaria que fossem definidas duas bandeiras de cartões de crédito: Visa e MasterCard. Para cada tipo de cartão, é necessário disponibilizar funções para **gerar um número de cartão** de crédito e **definir um limite de crédito**. Além disso, deve-se considerar a possibilidade de incluir novos tipos de cartões no sistema no futuro."*

O arquiteto da equipe, ao reconhecer as semelhanças no código e antecipando a possibilidade de adição de novas bandeiras de cartões, solicitou a criação de uma estrutura de herança para promover o reuso e a organização do software.

Para atender a esse requisito, você deve criar uma nova classe denominada *CreditCard* que deverá encapsular as características compartilhadas entre as diferentes bandeiras de cartão de crédito.

A classe deverá ser composta pelos seguintes atributos:

Atributo	Tipo	Contexto
ownerName	*Cadeira de Caracteres*	Nome do proprietário

bank	*IBank*	Banco
monthlyIncome	*double*	Salário do proprietário
limit	*double*	Limite do Cartão
number	*Cadeira de Caracteres*	Número do cartão
balance	*double*	Limite já utilizado

Para a construção do construtor desta classe, você precisará utilizar funções que serão definidas na própria classe. Portanto, começaremos pela implementação dos métodos e, em seguida, retornaremos para a definição do construtor.

A classe *CreditCard* deve conter dois métodos: *generateCreditCardNumber* e *grantCreditLimit*. Esses métodos são responsáveis por gerar um número para o cartão de crédito e conceder um limite de crédito para o cartão, respectivamente.

O método *generateCreditCardNumber* não requer parâmetros e deve ter um retorno do tipo *String*. Em outras palavras, quando chamado, ele deve retornar uma sequência de 16 dígitos aleatórios que representam o número do cartão de crédito. Cada vez que o método for invocado, ele deve gerar um número de cartão diferente. O número do cartão deve ser composto apenas por dígitos, sem espaços ou hifens, seguindo o formato abaixo como exemplo:

2891974975318974

O método *grantCreditLimit* deve receber um parâmetro do tipo *double* chamado *monthlyIncome*. Este método deve retornar o resultado da multiplicação do valor de *monthlyIncome* por três, ou seja, o valor do limite de crédito será igual a três vezes o salário mensal do titular do cartão.

Com essas funções definidas, prosseguiremos com a elaboração do construtor da classe *CreditCard*. O construtor da classe deve aceitar três parâmetros, nessa ordem: *ownerName*, *bank* e *monthlyIncome*. Esses parâmetros serão usados para inicializar os atributos da classe. Os demais atributos da classe devem ser inicializados conforme as seguintes especificações:

- **limit:** Este atributo será inicializado por meio da chamada da função *grantCreditLimit*, passando como parâmetro o valor de *monthlyIncome* recebido no construtor;
- **number:** Este atributo será inicializado por meio da chamada da função *generateCreditCardNumber*;
- **balance:** Deve ser inicializado com o valor 0.

6) Após a implementação da classe *CreditCard*, os clientes compartilharam uma nova regra de negócio relacionada aos cartões de crédito:

"É crucial que os métodos para gerar novos números de cartões e estabelecer limites para os clientes sigam regras específicas para cada tipo de cartão. Cartões Visa devem iniciar com o número 3 e ter o limite definido como o dobro do salário do proprietário do cartão, enquanto cartões MasterCard devem começar com o número 5 e ter o limite determinado como 2,5 vezes o salário do proprietário do cartão."

Começaremos a implementação com os cartões Visa. Para isso, você deve criar uma classe chamada Visa que herda os atributos e métodos da classe *CreditCard*. Essa classe não terá atributos adicionais, mas deve incluir um construtor que receba todos os atributos necessários para a superclasse (*CreditCard*) e chame o construtor da superclasse, passando esses atributos como argumentos.

De acordo com os requisitos especificados, a classe Visa deve incluir métodos próprios para gerar números de cartões de crédito (*generateCreditCardNumber*) e atribuir limites de crédito (*grantCreditLimit*) aos titulares dos cartões de crédito. Para atender a esses requisitos, você deve aplicar os conceitos de polimorfismo, realizando a sobrescrita de métodos.

O método *generateCreditCardNumber* é semelhante ao método correspondente criado na classe *CreditCard*. Ele deve gerar uma sequência de 16 dígitos no formato de String. No entanto, todos os cartões gerados por este método devem ter o número 3 como primeiro dígito, conforme especificado para os cartões Visa.

O método *grantCreditLimit* também é semelhante ao método correspondente na classe *CreditCard*. A diferença é que, para os cartões Visa, o limite deve ser definido multiplicando o salário (*monthlyIncome*) do proprietário por 2, de acordo com as regras estabelecidas.

7) Para dar continuidade ao nosso projeto, vamos criar um novo tipo de cartão de crédito. Crie a classe *MasterCard* que herda os atributos e métodos da classe *CreditCard*. Esta nova classe não terá novos atributos, mas deve incluir um construtor que aceite todos os atributos necessários da superclasse (*CreditCard*) e os repasse ao construtor da superclasse.

De acordo com os requisitos, a classe *MasterCard* deve implementar métodos próprios para gerar números de cartões de crédito (método *generateCreditCardNumber*) e definir limites de crédito (método *grantCreditLimit*) para os proprietários dos cartões. Isso requer a aplicação dos conceitos de polimorfismo por meio da sobrescrita de métodos.

O método *generateCreditCardNumber* na classe *MasterCard* é semelhante ao método correspondente na classe *CreditCard*, gerando uma sequência de 16 números no formato de String. A diferença é que todos os cartões gerados por este método devem começar com o número 5, conforme especificado para cartões MasterCard.

O método *grantCreditLimit* na classe MasterCard também é semelhante ao método correspondente na classe *CreditCard*, mas a diferença é que os cartões MasterCard terão seus limites definidos multiplicando o salário (*monthlyIncome*) do proprietário por 2,5, de acordo com as regras estabelecidas.