Exercício Prático Aula Prática 8 - 18/05/2017
Programação de Computadores 2 - Programação Orientada a Objetos
Polimorfismo
Prof. Daniel Hasan Dalip – hasan@decom.cefetmg.br

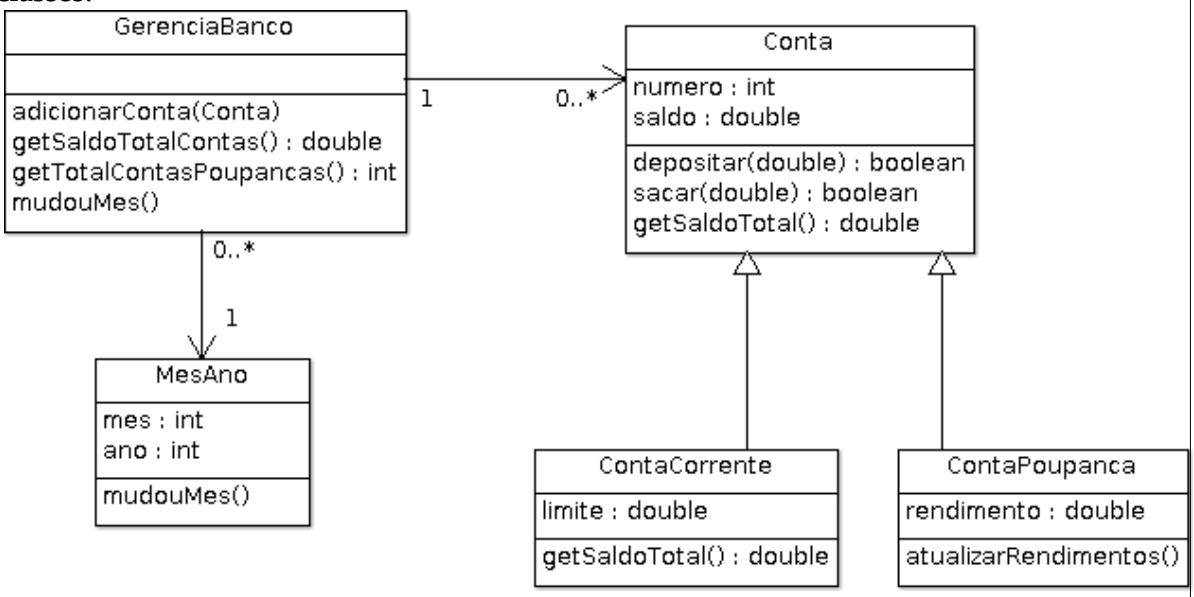
Crie um projeto para este exercício. Após finalizado o exercício, salve o projeto em um zip e envie-o pelo Moodle (Entrega Roteiro prática 6). Não será aceito o recebimento via e-mail. Plágio não será tolerado e tanto a pessoa que forneceu quanto a que utilizou perderão os pontos da prática.

Entrega:

até o final da aula: todos os métodos novos da classe conta, ContaCorrente e ContaPoupanca além da classe MesAno com todos seus métodos e métodos adicionarConta e getSaldoTotalContas da classe GerenciaBanco.

até dia 24/05: o restante

Exercício 1: Dado o exercício 1 da prática sobre Herança:
“Implemente as classes Conta, ContaCorrente e ContaPoupanca. A conta e ContaCorrente está descrita na aula sobre herança. A única diferença é que a classe Conta deverá ter um método setSaldo (para ser usado pela ContaPoupanca). Implemente a classe ContaPoupanca ela deverá ter o valor do rendimento de 0 até 1 representando o redimento de 0 a 100%. O rendimento, juntamente com o número da conta, será inicializado no construtor e deverá ter seus métodos de alteração. Além disso, crie o método atualizarRendimento que atualiza o saldo a partir do rendimento que a poupança possui. Por exemplo, se o saldo é de R$ 200,00 e o rendimento = 0.10 (10%), então após executar atualizarRendimento o saldo será de R$ 220,00.” Agora você deverá complementar este sistema, de acordo com este diagrama de classes:



Dessa forma, você deverá fazer os seguintes métodos/classes:

- **getSaldoTotal()**: Este método retorna o saldo total de uma conta. Ele não deverá possuir parâmetros. O cálculo do saldo total em uma conta (ou seja, em uma instância da classe Conta) é realizado apenas retornando seu saldo, enquanto isso, na ContaCorrente, é a soma do saldo mais o seu limite.

Assim, o método getSaldoTotal assume comportamentos diferentes dependendo do tipo de conta. Para implementarmos isso, deveremos usar **polimorfismo**. Ou seja, implementamos o método **getSaldoTotal** em **Conta** e também em **ContaCorrente**. Perceba que não precisamos mudar nada em **ContaPoupanca**, pois o cálculo do saldo total na **ContaPoupanca** é o mesmo que em **Conta**.
Coloque **@override** antes da assinatura de método da **ContaCorrente**. Ele indica que este método está reimplementando um método da superclasse. Ao fazer o teste número 2 você irá perceber a importância desta diretiva.

```
@Override
public double getSaldoTotal() {
        (...)
}
```

Essa diretiva, apesar de opcional, é importante. Se quisermos renomear o **getSaldoTotal** em **ContaCorrente**, devemos renomear também o **getSaldoTotal** na sua superclasse para que o funcionamento da classe sempre fique conforme o esperado. (Veja no segundo teste)

- Classe **MesAno**: esta classe será útil para definirmos qual mês e ano estamos. Por isso, ela deverá possuir dois atributos do tipo **int**: mês e ano. Além disso, esta classe possuirá dois construtores: um passando o mês e ano como parâmetro. No outro, o parâmetro é apenas o ano e o mês é definido dentro do construtor como janeiro. Use o **this** para chamar o outro construtor.
  Nesta classe, você implementará apenas os getters e, ao invés dos setters, faça um método mudouMes(). Este método sempre que chamado, passa para o próximo mês e, quando estiver em dezembro, passa para janeiro do próximo ano. Este método não deverá possuir parâmetros nem tipo de retorno. Implemente também o toString exibindo o mês e ano correspondente.

- Classe **GerenciaBanco**: esta classe terá uma (e apenas uma) lista de contas e um objeto do tipo **MesAno** para sabermos o mês atual. Tais contas podem ser conta-poupança, conta-corrente ou uma conta simples. A classe GerenciaBanco possuirá dois construtores: um que recebe como parâmetro o mês e ano atual (duas variáveis do tipo int) e, com isso, é instanciado o objeto do tipo MesAno. No outro construtor deverá ser passando como parâmetro um objeto do tipo MesAno.
  De acordo com o diagrama de classes deste sistema, um objeto da classe GerenciaBanco sempre deve sempre possuir um objeto MesAno associado a ele. Porém, perceba que por meio do construtor que passa o objeto MesAno como parâmetro, é possível passar um objeto nulo (Ou seja null) e, assim, o objeto GerenciaBanco não possuirá um objeto MesAno associado. Por isso, caso seja passado um objeto nulo, você deverá instanciar um objeto MesAno que inicie em janeiro de 2000. Dica: você pode usar a instrução **objMesAno == null** dentro de um **if** sem problema algum.
  Além disso, implemente os seguintes métodos:
  - **adicionarConta(Conta c)**: Adiciona uma conta bancária na lista de contas. Este método deve possuir apenas um parâmetro e tal parâmetro deve ser do tipo **Conta**. Crie a exceção ContaJaExistente. Essa exceção será lançada caso seja adicionado uma conta que já exista. Para descobrir se uma conta existe ou não, utilize o método **contains(Object o)** da class Lista. Porém, para que ele funcione corretamente, você deverá indicar na classe conta o que significa uma conta ser igual a outra. Assim, implemente o método “public boolean equals(Object o)” na classe Conta. Este método deve retornar verdadeiro caso o objeto, passado como parâmetro, seja igual ao objeto corrente. Uma conta é igual a outra caso possua o

mesmo número.
Curiosidade: Veja no javadoc do método contains na classe Lista: https://docs.oracle.com/javase/7/docs/api/java/util/List.html#contains(java.lang.Object). Neste JavaDoc ele explica o critério para que o contains retorne verdadeiro: considere que tenha sido passado como parâmetro o objeto **o.** O contains retornará verdadeiro se existir pelo menos um objeto **e** na lista no qual o.equals(e)==true , ou tanto o==null e e==null.

- **getSaldoTotalContas**: este método irá retornar o somatório do saldo total de todas as suas contas (independente do tipo). Note que, para isso, **não** é necessário fazer um **downcast** para as classes ContaCorrente e ContaPoupanca. No código, apenas invocamos o método getSaldoTotal de Conta e em tempo de execução é definido qual método será realmente executado (caso seja uma ContaCorrente, é executado o método getSaldoTotal de ContaCorrente). Este é um exemplo de **polimorfismo**. Este método não deverá possuir parâmetros.
  - Dica: para navegarmos pelas contas, você pode utilizar a seguinte instrução relativa ao **foreach** em java:

```
for(Conta c : lstContas){
   (…)
}
```

  onde lstContas é uma lista de contas.

- **getTotalContasPoupancas**: calcula e retorna a quantidade de contas-poupanças existentes. Para isso, use o **instanceof** para garantir que é uma conta-poupança (similar ao exemplo demonstrado na aula teórica). Este método não deverá possuir parâmetros.
- **getContasCorrentes**: retorna uma lista (classe List) com todas as contas correntes. Este método não deverá possuir parâmetros.
- **mudouMes**: este método não possui parâmetros e retorna nenhum elemento. Tal método, além de alterar a data para o mês seguinte (utilizando o seu objeto **MesAno)**, atualizará o saldo das contas-poupanças de acordo com o seu rendimento. Para isso, você pode usar o método atualizarRendimento da classe ContaPoupanca (implementado na prática anterior).
  Note que, para isso, você deve navegar por todas as contas, verificar se é uma ContaPoupanca e, caso seja uma ContaPoupanca, fazer um **cast** para ContaPoupanca e atualizar o seu rendimento.
  Note que o método mudouMes da classe GenciaBanco **não** está sobrepondo o mudouMes da classe MesAno, pois a relação entre GerenciaBanco e MesAno é uma **associação** e não **herança**.

- **toString**: O toString da classe GerenciaBanco deverá exibir o mês/ano atual, saldo total de contas e o número total de contas poupanças. Além disso, deverá exibir os dados da conta poupança de maior rendimento (e apenas ela, não é necessário exibir as demais).

Implemente os seguintes testes:
Crie uma classe Principal com o método main para implementar os seguintes testes:
1º Teste – utilização do polimorfismo:
Crie um objeto do tipo **GerenciaBanco** iniciando no mês de janeiro de 1984. Logo após adicione a ele três contas-correntes com limite de R$ 100,00, duas contas-poupanças: uma com rendimento de 0,70% e outra com rendimento de 0,80%; e duas contas simples. Para

cada uma dessas contas, faça um depósito de uma determinada quantia. Logo após, você deverá imprimir os dados do GerenciaBanco (usando o método toString do GerenciaBanco) e, em seguida, simular a mudança do mês com o objetivo chegar em janeiro de 1985 (i.e., passando-se 1 ano) e, logo após, passando mais 30 anos. Finalmente, exiba novamente os dados de GerenciaBanco.

2º Teste – importância da anotação "@override" antes do método:
Utilizando o teste anterior, escreva em um papel o saldo total final obtido. Perceba que, em ContaCorrente, se você alterar manualmente (i.e. sem usar refatoração) o nome do método getSaldoTotal, haverá erro de compilação. Isso ocorre, pois, ao usar a anotação override, você é obrigado a "sobrepor" um método de sua superclasse. Então, primeiramente, tire a anotação "@override" e altere manualmente o nome do metodo para getST(). Execute a classe principal novamente. Perceba que o saldo total das contas-correntes não serão o mesmo.

Isso ocorre, pois, ao mudar o nome de getSaldoTotal para getST, ao executarmos o método getSaldoTotal de Conta não é utilizado polimorfismo, pois ContaCorrente não possuirá seu respectivo método que calcula o saldo total.

Logo após este teste, coloque um comentário no main escrevendo o saldo total nas duas situações (renomeando ou não o método getSaldoTotal) e volte o método getST() para seu nome original.

3º Teste – relembrando vetores em java:
Na classe MesAno, você deverá criar um método getNomeMes() que exibe o nome do mês atual. Este método não possuirá parâmetro algum e deverá retornar o mês: janeiro, fevereiro, março, … de acordo com o seu atributo mês. Para simplificar esta implementação, você deve criar um vetor com todos os meses e utilizar o número do mês como índice de tal vetor. Por exemplo, seja **mes** o atributo que representa o valor numérico do mês, crie vetor: {"janeiro", "fevereiro",..., "dezembro"} e caso mes=1, você deverá retornar a posição 0 do vetor (no geral, você sempre irá resgatar a posição mes-1).

Ao declarar o vetor de meses, declare-o de tal que exista apenas um vetor durante toda a execução do programa. Ou seja este vetor será:

- uma variável do método **getNomeMes?**
- um atributo não *static* da classe **MesAno?**
- um atributo *static* da classe **MesAno?**

Outra pergunta: este vetor será uma variável ou uma constante (final)? Implemente da melhor forma possível.

Após implementada esta modificação, altere o toString para que exiba o nome do mês ao invés do número e realize o primeiro teste novamente.

Curiosidade: ao colocar uma referência como final, significa que a referência não poderá mudar (e não os valores internos desta referência). Por exemplo, se declararmos: "final MesAno objMesAno = new MesAno(1,2004);". Podemos executar: mês.mudouMes() sem problemas. Porém, não será possível atribuir uma outra referência à variável objMesAno. O similar ocorre com vetores: caso tivermos um vetor "final String[] arrMeses = {"x", "y", "z"}", não podemos atribuir outra referência ao vetor (ex: "arrMeses = new String[10]") mas podemos mudar os valores dentro deles (ex: arrMeses[0] = "z").