DESAFIO AULA 4

Java Platform

2019

Prof. Danilo Vitoriano
profdanilo.vitoriano@fiap.com.br

MBA FULL STACK DEVELOPER
MICROSERVICES, CLOUD & IoT

1. Classe Conta e TestaConta

1. Crie o projeto **contas**. Você pode usar o atalho control + n ou então ir no menu `File -> New -> Project... -> Java Project.`

2. Dentro do projeto **contas**, crie a classe Conta no pacote conta. Uma conta deve ter as seguintes informações: saldo (double), titular (String), numero (int) e agencia (String). Na classe Conta, crie os métodos deposita e saca adicionando e retirando do valor de saldo. Crie também um pacote main com uma classe TesteDaConta, com o método main e instancie uma conta.

2. Pacotes

1. Faça uma refatoração do nome dos pacotes para seguir o padrão Java:

`br.com.fiap.contas.main`: colocar a classe com o método main aqui (o Teste)
`br.com.fiap.contas.modelo`: colocar a classe Conta

3. Javadoc

1. Gere o Javadoc do seu sistema. Para isso, vá ao menu Project, depois à opção Generate Javadoc, se estiver na perspectiva Java. Se não, dê um clique com o botão direito no seu projeto, escolha Export e depois javadoc e siga o procedimento descrito na última seção deste capítulo.

4. Herança e Polimorfismo

1. Crie a classe ContaCorrente no pacote br.com.fiap.contas.modelo e faça com que ela seja filha da classe Conta

2. Crie a classe ContaPoupanca no pacote br.com.fiap.contas.modelo e faça com que ela seja filha da classe Conta

3. Criar um método getTipo em cada uma de nossas contas fazendo com que a conta corrente devolva a string "Conta Corrente" e a conta poupança devolva a string "Conta Poupança":

```
public class ContaCorrente extends Conta {
    public String getTipo() {
        return "Conta Corrente";
    }
}

public class ContaPoupanca extends Conta {
    public String getTipo() {
        return "Conta Poupança";
    }
}
```

4. Ao tentarmos chamar o método getTipo, o Eclipse reclamou que esse método não existe na classe Conta apesar de existir nas classes filhas. Como estamos tratando todas as contas genericamente, só conseguimos acessar os métodos da classe mãe. Vamos então colocá-lo na classe Conta:

```
public class Conta {
    public String getTipo() {
        return "Conta";
    }
}
```

5. Se algum dia precisarmos alterar o valor da taxa no saque, teríamos que mudar em todos os lugares onde fazemos uso do método saca. Esta lógica deveria estar encapsulada dentro

do método saca de cada conta. Vamos então sobrescrever o método dentro da classe ContaCorrente:

```
public class ContaCorrente extends Conta {
    @Override
    public void saca(double valor) {
        this.saldo -= (valor + 0.10);
    }

  // restante da classe
}
```

Repare que, para acessar o atributo saldo herdado da classe Conta, você vai precisar mudar o modificador de visibilidade de saldo para protected.

6. Rode a classe TestaContas, adicione uma conta de cada tipo e veja se o tipo é apresentado corretamente na lista de contas da tela inicial.

7. Vamos começar implementando o método transfere na classe Conta:

```
public void transfere(double valor, Conta conta) {
    this.saca(valor);
    conta.deposita(valor);
}
```

5. Classes Abstratas

1. Repare que a nossa classe Conta é uma excelente candidata para uma classe abstrata. Por quê? Que métodos seriam interessantes candidatos a serem abstratos?

Transforme a classe Conta em abstrata:

```
public abstract class Conta {
    // ...
}
```

2. Como a classe Conta agora é abstrata, não conseguimos dar new nela mais. Se não podemos dar new em Conta, qual é a utilidade de ter um método que recebe uma referência a Conta como argumento? Aliás, posso ter isso?

3. Apenas para entender melhor o abstract, comente o método getTipo() da ContaPoupanca, dessa forma ele herdará o método diretamente de Conta.

Transforme o método getTipo() da classe Conta em abstrato. Repare que, ao colocar a palavra chave abstract ao lado do método, o Eclipse rapidamente vai sugerir que você deve remover o corpo (body) do método com um quick fix.

Sua classe Conta deve ficar parecida com:

```
public abstract class Conta {
    // atributos e métodos que já existiam

    public abstract String getTipo();
}
```

4. Qual é o problema com a classe ContaPoupanca?

Descomente o método getTipo na classe ContaPoupanca, e se necessário altere-o para que a classe possa compilar normalmente.

6. Interfaces

1. Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso vamos criar uma interface no pacote br.com.fiap.contas.modelo do nosso projeto contas já existente:

```
public interface Tributavel {
  public double getValorImposto();
}
```

Lemos essa interface da seguinte maneira: "todos que quiserem ser tributável precisam saber retornar o valor do imposto, devolvendo um double".

Alguns bens são tributáveis e outros não, ContaPoupanca não é tributável, já para ContaCorrente você precisa pagar 1% da conta e o SeguroDeVida tem uma taxa fixa de 42 reais mais 2% do valor do seguro.

Aproveite o Eclipse! Quando você escrever implements Tributavel na classe ContaCorrente, o quick fix do Eclipse vai sugerir que você reescreva o método; escolha essa opção e, depois, preencha o corpo do método adequadamente:

```
public class ContaCorrente extends Conta implements Tributavel {

    // outros atributos e métodos

    public double getValorImposto() {
        return this.getSaldo() * 0.01;
    }
}
```

Crie a classe SeguroDeVida, aproveitando novamente do Eclipse, para obter:

```
public class SeguroDeVida implements Tributavel {
    private double valor;
    private String titular;
    private int numeroApolice;
```

```
    public double getValorImposto() {
        return 42 + this.valor * 0.02;
    }

// getters e setters para os atributos
}
```

Além disso, escreva o método getTipo para que o tipo do produto apareça na interface gráfica:

```
public String getTipo(){
    return "Seguro de Vida";
}
```

2. Execute a classe TestaContas e tente cadastrar um novo seguro de vida. O seguro cadastrado deve aparecer na tabela de seguros de vida.

3. (opcional) Crie a classe TestaTributavel com um método main para testar o nosso exemplo:

```
public class TestaTributavel {

    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente();
        cc.deposita(100);
        System.out.println(cc.getValorImposto());

        // testando polimorfismo:
        Tributavel t = cc;
        System.out.println(t.getValorImposto());
    }
}
```

7. Exceções

1. Na classe Conta, modifique o método deposita(double x): Ele deve lançar uma exception chamada IllegalArgumentException, que já faz parte da biblioteca do Java, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo).

```
 public void deposita(double valor) {
     if (valor < 0) {
         throw new IllegalArgumentException();
     } else {
         this.saldo += valor;
     }
 }
```

2. Rode a aplicação, cadastre uma conta e tente depositar um valor negativo. O que acontece?

3. Ao lançar a IllegalArgumentException, passe via construtor uma mensagem a ser exibida. Lembre que a String recebida como parâmetro é acessível depois via o método getMessage() herdado por todas as Exceptions.

```
 public void deposita(double valor) {
     if (valor < 0) {
         throw new IllegalArgumentException("Você tentou
depositar" +
                                            " um valor
negativo");
     } else {
         this.saldo += valor;
     }
 }
```

Rode a aplicação novamente e veja que agora a mensagem aparece na tela.

4. Faça o mesmo para o método saca da classe ContaCorrente, afinal o cliente também não pode sacar um valor negativo!

5. Vamos validar também que o cliente não pode sacar um valor maior do que o saldo disponível em conta. Crie sua própria Exception, SaldoInsuficienteException. Para isso, você precisa criar uma classe com esse nome que seja filha de RuntimeException.

```
 public class SaldoInsuficienteException extends RuntimeException
{

 }
```

No método saca da classe ContaCorrente vamos utilizar esta nova Exception:

```
 @Override
 public void saca(double valor) {
     if (valor < 0) {
         throw new IllegalArgumentException("Você tentou sacar um
valor negativo");
     }
     if (this.saldo < valor) {
         throw new SaldoInsuficienteException();
     }
     this.saldo -= (valor + 0.10);
 }
```

Atenção: nem sempre é interessante criarmos um novo tipo de exception! Depende do caso. Neste aqui, seria melhor ainda utilizarmos IllegalArgumentException. A boa prática diz que devemos preferir usar as já existentes do Java sempre que possível.

6. (opcional) Coloque um construtor na classe SaldoInsuficienteException que receba o valor que ele tentou sacar (isto é, ele vai receber um double valor).

Quando estendemos uma classe, não herdamos seus construtores, mas podemos acessá-los através da palavra chave super de dentro de um construtor. As exceções do Java possuem uma série de construtores úteis para poder populá-las já com uma mensagem de erro. Então vamos criar um construtor em SaldoInsuficienteException que delegue para o construtor de sua mãe. Essa vai guardar essa mensagem para poder mostrá-la ao ser invocado o método getMessage:

```
 public class SaldoInsuficienteException extends RuntimeException
{
```

```
    public SaldoInsuficienteException(double valor) {
        super("Saldo insuficiente para sacar o valor de: " +
valor);
    }
 }
```

Dessa maneira, na hora de dar o throw new SaldoInsuficienteException você vai precisar passar esse valor como argumento:

```
 if (this.saldo < valor) {
    throw new SaldoInsuficienteException(valor);
 }
```

Atenção: você pode se aproveitar do Eclipse para isso: comece já passando o valor como argumento para o construtor da exception e o Eclipse vai reclamar que não existe tal construtor. O quick fix (ctrl + 1) vai sugerir que ele seja construindo, poupando-lhe tempo!

E agora, como fica o método saca da classe ContaCorrente?

7. (opcional) Declare a classe SaldoInsuficienteException como filha direta de Exception em vez de RuntimeException. Ela passa a ser checked. O que isso resulta?

Você vai precisar avisar que o seu método saca() throws SaldoInsuficienteException, pois ela é uma checked exception. Além disso, quem chama esse método vai precisar tomar uma decisão entre try-catch ou throws. Faça uso do quick fix do Eclipse novamente!

Depois, retorne a exception para unchecked, isto é, para ser filha de RuntimeException, pois utilizaremos ela assim em exercícios dos capítulos posteriores.

8. java.lang.Object

1. Reescreva o método toString da sua classe Conta fazendo com que uma mensagem mais explicativa seja devolvida. Lembre-se de aproveitar dos recursos do Eclipse para isto: digitando apenas o começo do nome do método a ser reescrito e pressionando ctrl + espaço, ele vai sugerir reescrever o método, poupando o trabalho de escrever a assinatura do método e cometer algum engano.

```
public abstract class Conta {

  protected double saldo;

    @Override
    public String toString() {
        return "[titular=" + titular + ", numero=" + numero
           + ", agencia=" + agencia + "]";
    }
  // restante da classe

}
```

Rode a aplicação novamente, cadastre duas contas e verifique que aconteceu.

2. Reescreva o método equals da classe Conta para que duas contas com o mesmo número e agência sejam consideradas iguais. Esboço:

```
public abstract class Conta {

    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }

        Conta outraConta = (Conta) obj;

        return this.numero == outraConta.numero &&
            this.agencia.equals(outraConta.agencia);
```

```
    }
}
```

Você pode usar o ctrl + espaço do Eclipse para escrever o esqueleto do método equals, basta digitar dentro da classe equ e pressionar ctrl + espaço.

9. java.lang.String

1. Teste os exemplos desse capítulo, para ver que uma String é imutável. Por exemplo, cria a classe TestaString:

```
public class TestaString {

    public static void main(String[] args) {
        String s = "fj11";
        s.replaceAll("1", "2");
        System.out.println(s);
    }

}
```

Como fazer para ele imprimir fj22?

10. Arrays

Para consolidarmos os conceitos sobre arrays, vamos fazer alguns exercícios que não interferem em nosso projeto.

1. Crie uma classe TestaArrays e no método main crie um array de contas de tamanho 10. Em seguida, faça um laço para criar 10 contas com saldos distintos e colocá-las no array. Por exemplo, você pode utilizar o índice do laço e multiplicá-lo por 100 para gerar o saldo de cada conta:

```
Conta[] contas = new Conta[10];

for (int i = 0; i < contas.length; i++) {
  Conta conta = new ContaCorrente();
  conta.deposita(i * 100.0);
  // escreva o código para guardar a conta na posição i do array
}
```

2. Ainda na classe TestaArrays, faça um outro laço para calcular e imprimir a média dos saldos de todas as contas do array.

3. (opcional) Crie uma classe Banco dentro do pacote br.com.fiap.contas.modelo O Banco deve ter um nome e um número (obrigatoriamente) e uma referência a uma array de Conta de tamanho 10, além de outros atributos que você julgar necessário.

```
public class Banco {
   private String nome;
   private int numero;
   private Conta[] contas;

   // outros atributos que você achar necessário

   public Banco(String nome, int numero) {
       this.nome = nome;
       this.numero = numero;
       this.contas = new ContaCorrente[10];
   }
```

```
    // getters para nome e número, não colocar os setters pois já
recebemos no
    // construtor
}
```

5. (opcional) A classe Banco deve ter um método adiciona, que recebe uma referência a Conta como argumento e guarda essa conta.

Você deve inserir a Conta em uma posição da array que esteja livre. Existem várias maneiras para você fazer isso: guardar um contador para indicar qual a próxima posição vazia ou procurar por uma posição vazia toda vez. O que seria mais interessante?

Se quiser verificar qual a primeira posição vazia (nula) e adicionar nela, poderia ser feito algo como:

```
public void adiciona(Conta c) {
    for(int i = 0; i < this.contas.length; i++){
        // verificar se a posição está vazia
        // adicionar no array
    }
}
```

É importante reparar que o método adiciona não recebe titular, agencia, saldo, etc. Essa seria uma maneira nem um pouco estruturada, muito menos orientada a objetos de se trabalhar. Você antes cria uma Conta e já passa a referência dela, que dentro do objeto possui titular, saldo, etc.

6. (opcional) Crie uma classe TestaBanco que possuirá um método main. Dentro dele crie algumas instâncias de Conta e passe para o banco pelo método adiciona.

```
Banco banco = new Banco("CaelumBank", 999);
//     ....
```

Crie algumas contas e passe como argumento para o adiciona do banco:

```
ContaCorrente c1 = new ContaCorrente();
c1.setTitular("Batman");
c1.setNumero(1);
```

```
c1.setAgencia(1000);
c1.deposita(100000);
banco.adiciona(c1);

ContaPoupanca c2 = new ContaPoupanca();
c2.setTitular("Coringa");
c2.setNumero(2);
c2.setAgencia(1000);
c2.deposita(890000);
banco.adiciona(c2);
```

Você pode criar essas contas dentro de um loop e dar a cada um deles valores diferentes de depósitos:

```
for (int i = 0; i < 5; i++) {
   ContaCorrente conta = new ContaCorrente();
   conta.setTitular("Titular " + i);
   conta.setNumero(i);
   conta.setAgencia(1000);
   conta.deposita(i * 1000);
   banco.adiciona(conta);
}
```

Repare que temos de instanciar ContaCorrente dentro do laço. Se a instanciação de ContaCorrente ficasse acima do laço, estaríamos adicionado cinco vezes a mesma instância de ContaCorrente neste Banco e apenas mudando seu depósito a cada iteração, que nesse caso não é o efeito desejado.

7. (opcional) Percorra o atributo contas da sua instância de Banco e imprima os dados de todas as suas contas. Para fazer isso, você pode criar um método chamado mostraContas dentro da classe Banco:

```
public void mostraContas() {
   for (int i = 0; i < this.contas.length; i++) {
       System.out.println("Conta na posição " + i);
       // preencher para mostrar outras informacoes da conta
   }
}
```

Aí, através do seu main, depois de adicionar algumas contas, basta fazer:

```
banco.mostraContas();
```

Referências:

Apostila Caelum - Java Orientação a Objetos
Apostila J2SE - Wincomp