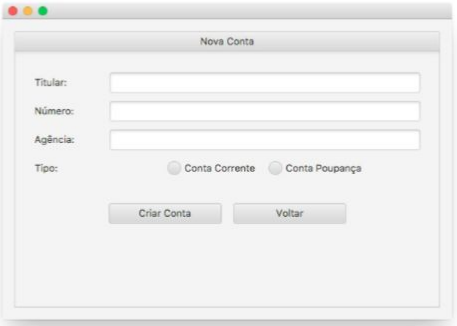
9.7 EXERCÍCIOS: HERANÇA E POLIMORFISMO

1. Vamos ter mais de um tipo de conta no nosso sistema então vamos precisar de uma nova tela para cadastrar os diferentes tipos de conta. Essa tela já está pronta e para utilizá-la só precisamos alterar a classe que estamos chamando no método `main()` no `TestaContas.java` :

```
package br.com.caelum.contas.main;

import br.com.caelum.javafx.api.main.SistemaBancario;

public class TestaContas {

    public static void main(String[] args) {
        SistemaBancario.mostraTela(false);
        // TelaDeContas.main(args);
    }
}
```

2. Ao rodar a classe `TestaContas` agora, teremos a tela abaixo:

Vamos entrar na tela de criação de contas para vermos o que precisamos implementar para que o sistema funcione. Para isso, clique no botão **Nova Conta**. A seguinte tela aparecerá:



Podemos perceber que além das informações que já tínhamos na conta, temos agora o tipo: se queremos uma conta corrente ou uma conta poupança. Vamos então criar as classes correspondentes.

- Crie a classe `ContaCorrente` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`
- Crie a classe `ContaPoupanca` no pacote `br.com.caelum.contas.modelo` e faça com que ela seja filha da classe `Conta`

3. Precisamos pegar os dados da tela para conseguirmos criar a conta correspondente. No `ManipuladorDeContas` vamos alterar o método `criaConta` . Atualmente, apenas criamos uma nova conta com os dados direto no código. Vamos fazer com que agora os dados sejam recuperados da tela para colocarmos na nova conta, faremos isso utilizando o objeto `evento` :

```
public void criaConta(Evento evento) {
   this.conta = new Conta();
   this.conta.setAgencia(evento.getString("agencia"));
   this.conta.setNumero(evento.getInt("numero"));
   this.conta.setTitular(evento.getString("titular"));
}
```

Mas precisamos dizer qual tipo de conta que queremos criar! Devemos então recuperar o tipo da conta escolhido e criar a conta correspondente. Para isso, ao invés de criar um objeto do tipo 'Conta', vamos usar o método `getSelecionadoNoRadio` do objeto `evento` para pegar o tipo, fazer um `if` para verificar esse tipo e só depois criar o objeto do tipo correspondente. Após essas mudanças, o método `criaConta` ficará como abaixo:

```
public void criaConta(Evento evento) {
   String tipo = evento.getSelecionadoNoRadio("tipo");
   if (tipo.equals("Conta Corrente")) {
       this.conta = new ContaCorrente();
   } else if (tipo.equals("Conta Poupança")) {
       this.conta = new ContaPoupanca();
   }
   this.conta.setAgencia(evento.getString("agencia"));
   this.conta.setNumero(evento.getInt("numero"));
   this.conta.setTitular(evento.getString("titular"));
}
```

4. Apesar de já conseguirmos criar os dois tipos de contas, nossa lista não consegue exibir o tipo de cada conta na lista da tela inicial. Para resolver isso, podemos 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";
    }
}
```

5. Altere os métodos `saca` e `deposita` para buscarem o campo `valorOperacao` ao invés de apenas `valor` na classe `ManipuladorDeContas`.

6. Vamos mudar o comportamento da operação de saque de acordo com o tipo de conta que estiver sendo utilizada. Na classe `ManipuladorDeContas` vamos alterar o método `saca` para tirar 10 centavos de cada saque em uma conta corrente:

```
public void saca(Evento evento) {
    double valor = evento.getDouble("valorOperacao");
    if (this.conta.getTipo().equals("Conta Corrente")){
        this.conta.saca(valor + 0.10);
    } else {
        this.conta.saca(valor);
```

```
    }
}
```

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";
    }
}
```

7. Agora o código compila mas temos um outro problema. A lógica do nosso saque vazou para a classe `ManipuladorDeContas`. 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`**.

Agora que a lógica está encapsulada, podemos corrigir o método `saca` da classe `ManipuladorDeContas`:

```
public void saca(Evento evento) {
    double valor = evento.getDouble("valorOperacao");
    this.conta.saca(valor);
}
```

Perceba que agora tratamos a conta de forma genérica!

8. 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.

   Agora, clique na conta corrente apresentada na lista para abrir a tela de detalhes de contas. Teste as operações de saque e depósito e perceba que a conta apresenta o comportamento de uma conta corrente conforme o esperado.

   E se tentarmos realizar uma transferência da conta corrente para a conta poupança? O que acontece?

9. Vamos começar implementando o método `transfere` na classe `Conta`:

```
public void transfere(double valor, Conta conta) {
```

```
        this.saca(valor);
        conta.deposita(valor);
    }
```

Também precisamos implementar o método `transfere` na classe `ManipuladorDeContas` para fazer o vínculo entre a tela e a classe `Conta`:

```
    public void transfere(Evento evento) {
        Conta destino = (Conta) evento.getSelecionadoNoCombo("destino");
        conta.transfere(evento.getDouble("valorTransferencia"), destino);
    }
```

Rode de novo a aplicação e teste a operação de transferência.

10. Considere o código abaixo:

```
Conta c = new Conta();
ContaCorrente cc = new ContaCorrente();
ContaPoupanca cp = new ContaPoupanca();
```

Se mudarmos esse código para:

```
Conta c = new Conta();
Conta cc = new ContaCorrente();
Conta cp = new ContaPoupanca();
```

Compila? Roda? O que muda? Qual é a utilidade disso? Realmente, essa não é a maneira mais útil do polimorfismo. Porém existe uma utilidade de declararmos uma variável de um tipo menos específico do que o objeto realmente é, como fazemos na classe `ManipuladorDeContas`.

É **extremamente importante** perceber que não importa como nos referimos a um objeto, o método que será invocado é sempre o mesmo! A JVM vai descobrir em tempo de execução qual deve ser invocado, dependendo de que tipo é aquele objeto, não importando como nos referimos a ele.

11. (Opcional) A nossa classe `Conta` devolve a palavra "Conta" no método `getTipo`. Use a palavra chave `super` nos métodos `getTipo` reescritos nas classes filhas, para não ter de reescreve a palavra "Conta" ao devolver os tipos "Conta Corrente" e "Conta Poupança".

12. (Opcional) Se você precisasse criar uma classe `ContaInvestimento`, e seu método `saca` fosse complicadíssimo, você precisaria alterar a classe `ManipuladorDeContas`?