



Aula 03

Programação Orientada à Objetos I
Prof. Me. Joseffe Barroso de Oliveira

UNIP - Universidade Paulista
Análise e Desenvolvimento de Sistemas

Herança

Imagine agora que nosso banco realize depósitos e saques de acordo com o tipo da conta. Se a conta for poupança, o cliente deve pagar 0.10 por saque. Se a conta for corrente, não há taxa.

Para implementar essa regra de negócio, vamos colocar um if no método Saca:

```
public void Saca(double valor)
{
    if(Tipo == ??????????)
    {
        this.Saldo -= valor + 0.10;
    }
    else
    {
        this.Saldo -= valor;
    }
}
```

Podemos criar um atributo na Conta, que especifica o tipo da conta como, por exemplo, um inteiro qualquer onde o número 1 representaria "conta poupança" e 2 "conta corrente".

A implementação seria algo como:

```
public class Conta
{
    public int Numero { get; set; }
    public double Saldo { get; private set; }

    public Cliente Titular { get; set; }

    public int Tipo { get; set; }

    public void Saca(double valor)
    {
        if(Tipo == 1)
        {
            this.Saldo -= valor + 0.10;
        }
        else
        {
            this.Saldo -= valor;
        }
    }

    public void Deposita(double valor)
    {
        this.Saldo += valor;
    }
}
```

Veja que uma simples regra de negócio como essa fez nosso código crescer muito. E poderia ser pior: imagine se nosso banco tivesse 10 tipos de contas diferentes. Esse if seria maior ainda.

Precisamos encontrar uma maneira de fazer com que a criação de novos tipos de conta não implique em um aumento de complexidade. Uma solução seria ter classes separadas para Conta (que é a corrente) e ContaPoupanca:

```
public class Conta
{
    public int Numero { get; set; }
    public double Saldo { get; private set; }

    public Cliente Titular { get; set; }

    public void Saca(double valor)
    {
        this.Saldo -= valor;
    }

    public void Deposita(double valor)
    {
        this.Saldo += valor;
    }
}

public class ContaPoupanca
{
    public int Numero { get; set; }
    public double Saldo { get; private set; }

    public Cliente Titular { get; set; }

    public void Saca(double valor)
    {
        this.Saldo -= (valor + 0.10);
    }

    public void Deposita(double valor)
    {
        this.Saldo += valor;
    }
}
```

Ambas as classes possuem código bem simples, mas agora o problema é outro: a repetição de código entre ambas as classes. Se amanhã precisarmos guardar "CPF", por exemplo, precisaremos mexer em todas as classes que representam uma conta no sistema. Isso pode ser trabalhoso.

A ideia é então reaproveitar código. Veja que, no fim, uma ContaPoupanca é uma Conta, pois ambos tem Numero, Saldo e Titular. A única diferença é o comportamento no momento do saque. Podemos falar que uma ContaPoupanca é uma Conta:

```
public class ContaPoupanca : Conta
{
    }
```

```
}
```

Quando uma classe é definida com o `:`, dizemos que ela herda da outra (Conta), ela ganha tudo o que a classe tem, como atributos e métodos. Por exemplo, se `ContaPoupanca` herdar de `Conta`, isso quer dizer que ela terá `Numero`, `Saldo`, `Titular`, `Saca()` e `Deposita()` automaticamente, sem precisar fazer nada. Veja o código abaixo:

```
public class ContaPoupanca : Conta
{
}

ContaPoupanca c = new ContaPoupanca();
c.Deposita(100.0);
```

Basta usar a notação `:` e o C# automaticamente herda os métodos e atributos da classe-pai. Mas a `ContaPoupanca` tem o comportamento de `Saca()` diferente. Para isso, basta reescrever o comportamento na classe filho, usando a palavra `override` e mudando a classe pai para indicar que o método pode ser sobrescrito (`virtual`):

```
public class Conta
{
    public virtual void Saca(double valor)
    {
        this.Saldo -= valor;
    }
    // ...
}

public class ContaPoupanca : Conta
{
    public override void Saca(double valor)
    {
        this.Saldo -= (valor + 0.10);
    }
}

ContaPoupanca c = new ContaPoupanca();
c.Deposita(100.0); // chama o comportamento escrito no pai
c.Saca(100.0); // chama o comportamento escrito na própria classe
```

Veja no código acima que invocamos tanto `Deposita()` quanto `Saca()`. No depósito, como a classe filha não redefiniu o comportamento, ele usa o do pai, o comportamento escrito na classe-pai será utilizado.

Já no saque, o comportamento usado é o que foi sobrescrito na classe filho.

Mas o código acima ainda não compila. Repare que o método `Saca()` da classe filho manipula o `Saldo`. Mas `Saldo` é privado! Atributos privados só são visíveis pela própria classe. Os filhos não enxergam.

Queremos proteger nosso atributo mas não deixá-lo privado nem público. Queremos proteger o suficiente para ninguém de fora acessar, mas quem herda ter acesso. Para resolver, alteraremos o modificador de acesso para

protected. Atributos/métodos marcados como protected são visíveis apenas para a própria classe e para as classes filhas:

```
public class Conta
{
    public int Numero { get; set; }
    public double Saldo { get; protected set; }

    // ...
}
```

A classe Conta ainda pode ser instanciada sem problemas:

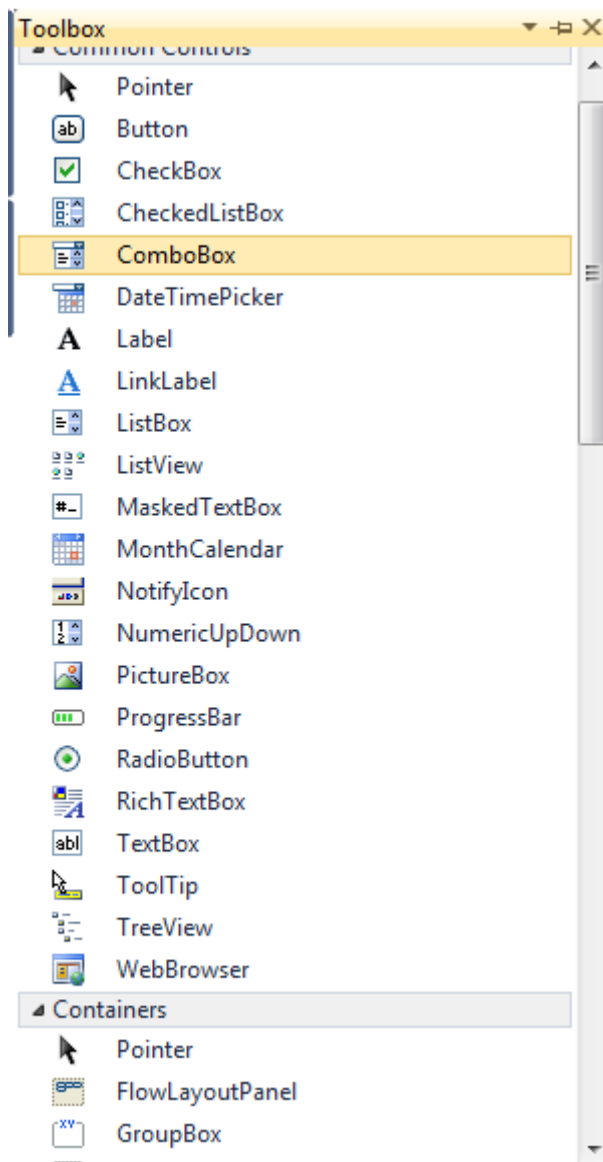
```
Conta c = new Conta();
c.Deposita(100.0);
```

Veja que com herança conseguimos simplificar e reutilizar código ao mesmo tempo. A herança é um mecanismo poderoso mas deve ser utilizado com parcimônia!

Combobox

Vamos fazer com que o caixa eletrônico consiga gerenciar múltiplas contas utilizando um novo componente do Windows Form conhecido como combo box.

Para adicionar um combo box no formulário, precisamos apenas abrir a janela **Toolbox** (Ctrl+W, X) e arrastar o combo box para dentro do formulário.



Para inserir os elementos que serão exibidos no combo box, precisaremos de uma variável que guarda a referência para o componente. Assim como no campo de texto, podemos definir o nome dessa variável através da janela properties.

Para acessar a janela properties do combo box, clique com o botão direito do mouse no combo box e selecione a opção `Properties`.

Dentro da janela properties, utilizaremos novamente o campo `(Name)` para definir o nome da variável que guardará a referência para o combo box.

Vamos utilizar `comboContas`.

Como agora estamos interessados em guardar múltiplas contas, colocaremos um array de contas como atributo na classe `Form1`:

```
public partial class Form1 : Form {  
    Conta [] contas;  
}
```

Agora, no método de inicialização do formulário, o `Form1_Load`, queremos criar as novas contas:

```
Conta contaDoVictor = new Conta();
contaDoVictor.Titular = "Victor";
contaDoVictor.Numero = 1;

Conta contaDoMario = new Conta();
contaDoMario.Titular = "Mario";
contaDoMario.Numero = 2;
```

Guardá-las no array:

```
this.contas = new Conta[2];
this.contas[0] = contaDoVictor;
this.contas[1] = contaDoMario;
```

E depois mostrar os titulares das contas como itens do combo box. Para adicionar um novo item no combo box, utilizamos o seguinte código:

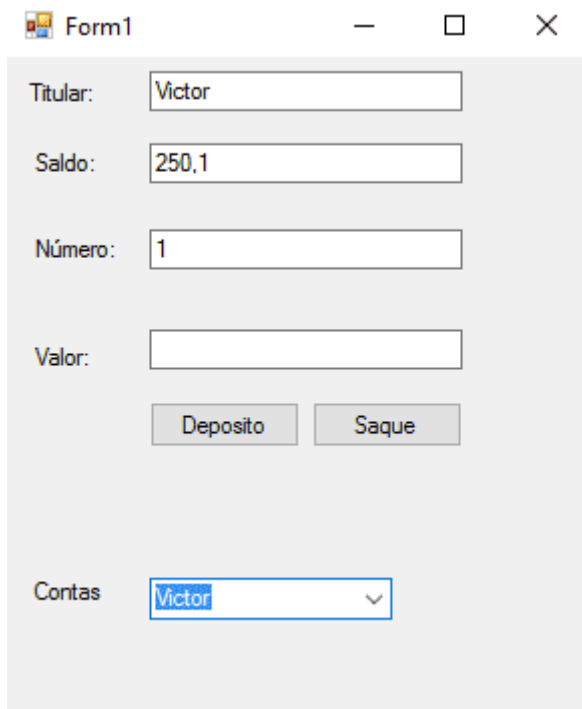
```
comboContas.Items.Add("Texto que aparecerá no combo box");
```

Logo, para mostrar os titulares como itens do combo box, utilizamos o seguinte código;

```
comboContas.Items.Add(contas[0].Titular);
comboContas.Items.Add(contas[1].Titular);
```

Ou podemos utilizar um `foreach`:

```
foreach(Conta conta in contas)
{
    comboContas.Items.Add(conta.Titular);
}
```



Agora que já conseguimos mostrar o combo box, queremos que a escolha de uma opção no combo, faça com que o formulário mostre a conta do titular selecionado.

Para associar uma ação ao evento de mudança de seleção do combo, precisamos apenas dar um duplo clique no combo box. Isso criará um novo método na classe `Form1`:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
}
```

Podemos recuperar qual é o índice (começando de zero) do item que foi selecionado pelo usuário lendo a

propriedade `SelectedIndex` do `comboContas`:

```
int indiceSelecionado = comboContas.SelectedIndex;
```

Esse índice representa qual é o elemento do array de contas que foi selecionado, logo podemos usá-lo para recuperar a conta que foi escolhida:

```
Conta contaSelecionada = contas[indiceSelecionado];
```

Depois de descobrir qual é a conta escolhida, vamos mostrá-la no formulário:

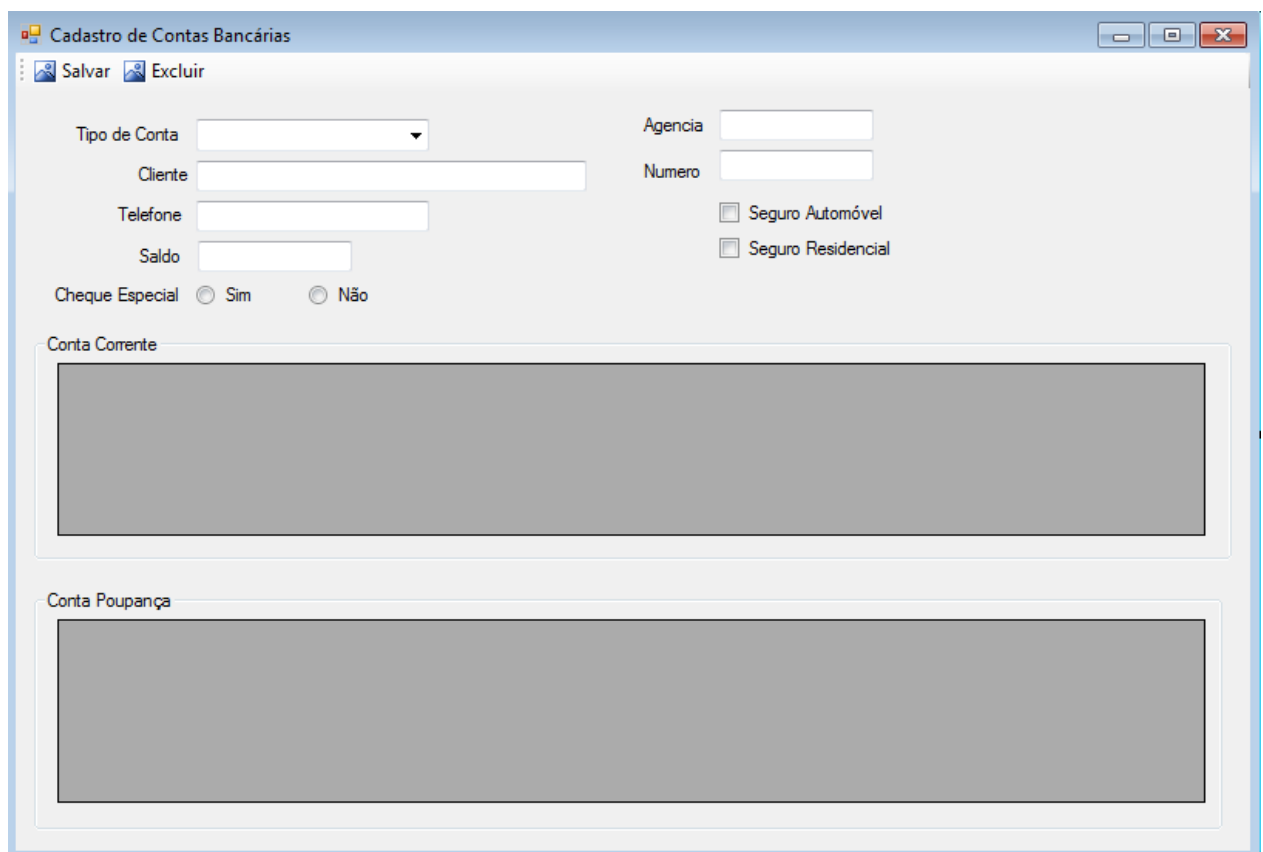
```
textoTitular.Text = contaSelecionada.Titular;
textoSaldo.Text = Convert.ToString(contaSelecionada.Saldo);
textoNumero.Text = Convert.ToString(contaSelecionada.Numero);
```


O código completo do `comboBox1_SelectedIndexChanged` fica da seguinte forma:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    int indiceSelecionado = comboContas.SelectedIndex;
    Conta contaSelecionada = contas[indiceSelecionado];
    textoTitular.Text = contaSelecionada.Titular;
    textoSaldo.Text = Convert.ToString(contaSelecionada.Saldo);
    textoNumero.Text = Convert.ToString(contaSelecionada.Numero);
}
```

Exercício

Utilizando os conceitos aprendidos em nossa aula e o projeto de exemplo, vamos desenvolver o seguinte programa:



A imagem mostra uma interface de usuário para um sistema de "Cadastro de Contas Bancárias". A janela possui uma barra de título com o ícone de uma pasta e o texto "Cadastro de Contas Bancárias". Abaixo da barra de título, há uma barra de menu com os itens "Salvar" e "Excluir".

O formulário principal contém os seguintes campos e controles:

- Tipo de Conta:** Um menu suspenso.
- Agencia:** Um campo de texto.
- Cliente:** Um campo de texto.
- Numero:** Um campo de texto.
- Telefone:** Um campo de texto.
- Saldo:** Um campo de texto.
- Seguro Automóvel:** Um botão de opção desativado.
- Seguro Residencial:** Um botão de opção desativado.
- Cheque Especial:** Um grupo de botões de opção com "Sim" selecionado e "Não" desativado.

Abaixo dos campos, há duas seções para o detalhamento da conta:

- Conta Corrente:** Uma caixa de texto grande e vazia.
- Conta Poupança:** Uma caixa de texto grande e vazia.

Referências Bibliográficas

www.alura.com.br