Unidade 5 3
5 Orientação a Objetos – herança, reescrita e polimorfismo 3
5.1 - Repetindo código? 3
5.2 - Reescrita de método 6
5.3 - Invocando o método reescrito 8
5.4 - Polimorfismo 9
5.5 - Um outro exemplo 11
5.6 - Um pouco mais... 13

Unidade 5

5 Orientação a Objetos – herança, reescrita e polimorfismo

Ao término desse capítulo, você será capaz de:

- Dizer o que é herança e quando utilizá-la;
- Reutilizar código escrito anteriormente;
- Criar classes filhas e reescrever métodos;
- Usar todo o poder que o polimorfismo oferece.

5.1 - Repetindo código?

Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe Funcionario:

```
class Funcionario
{
        String nome;
        String cpf;
        double salario;
        // métodos devem vir aqui
}
```

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco:

```
class Gerente
{
        String nome;
        String cpf;
        double salario;
        int senha;
        public bool Autenticar(int senha)
        {
            if (this.senha == senha)
            {
                Console.WriteLine("Acesso Permitido!");
                return true;
            }
            else
            {
                Console.WriteLine("Acesso Negado!");
                return false;
            }
        }
```

```
}
```

Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe Funcionario mais genérica, mantendo nela senha de acesso.

Caso o funcionário não fosse um gerente, deixaríamos este atributo vazio.

Essa é uma possibilidade. Mas e em relação aos métodos? A classe Gerente tem o método autenticar, que não faz sentido existir em um funcionário que não é gerente.

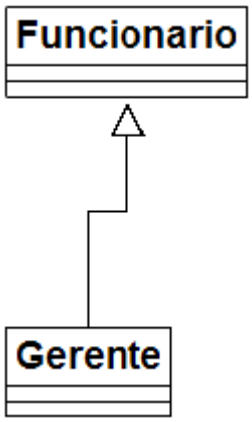
Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente!

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizar as informações principais do funcionário em um único lugar!

Existe um jeito, em C#, de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que a outra tem. Isto é uma relação de classe mãe e classe filha. No nosso caso, gostaríamos de fazer com que o Gerente tivesse tudo que um Funcionario tem, gostaríamos que ela fosse uma **extensão** de Funcionario. Fazemos isto através do símbolo “**:**”.

```
class Gerente : Funcionario
{
        int senha;
        public bool Autenticar(int senha)
        {
            if (this.senha == senha)
            {
                Console.WriteLine("Acesso Permitido!");
                return true;
            }
            else
            {
                Console.WriteLine("Acesso Negado!");
                return false;
            }
        }
}
```

Em todo momento que criarmos um objeto do tipo Gerente, este objeto possuirá também os atributos definidos na classe Funcionario, pois agora um Gerente **é um** Funcionario:




```
class TestaGerente
{
        static void Main(String[] args)
        {
            Gerente gerente = new Gerente();
            gerente.Nome = "João da Silva";
            gerente.Senha = 4231;
        }
}
```

Dizemos que a classe Gerente **herda** todos os atributos e métodos da classe mãe, no nosso caso, a Funcionario. Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessá-los diretamente.

Super e Sub classe

A nomenclatura mais encontrada é que Funcionario é a **superclasse** de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente é um Funcionário.

E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de Funcionario, public, pois dessa maneira qualquer um poderia alterar os atributos dos objetos deste tipo. Existe um outro modificador de acesso, o protected, que fica entre o private e o public. Um atributo protected só pode ser acessado (visível) pela própria classe ou suas subclasses.

```
class Funcionario
{
        protected String nome;
        protected String cpf;
        protected double salario;
        // métodos devem vir aqui
}
```

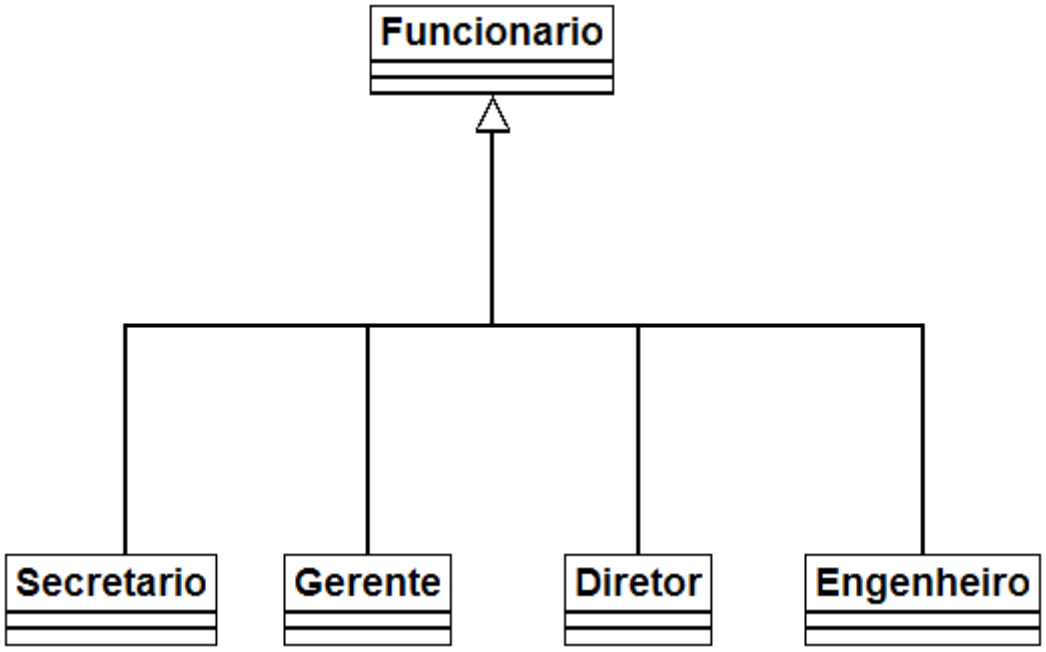
Sempre usar protected?

Então porque usar private? Depois de um tempo programando orientado a objetos, você vai começar a sentir que nem sempre é uma boa idéia deixar que a classe filha acesse os atributos da classe mãe, pois isso quebra um pouco a idéia de que só aquela classe deveria manipular seus atributos. Essa é uma discussão um pouco mais avançada.

Da mesma maneira podemos ter uma classe Diretor que estenda Gerente, e a classe Presidente pode estender diretamente de Funcionario.

Fique claro que essa é uma decisão de negócio. Se você vai estender Diretor de Gerente ou não, vai depender se Diretor "é um" Gerente.

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do C#.



5.2 - Reescrita de método

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

Vamos ver como fica a classe Funcionario:

```
class Funcionario
{
        protected String nome;
        protected String cpf;
        protected double salario;

        public double BuscarBonificacao()
        {
            return this.salario * 0.10;
        }
        // métodos
}
```

Se deixarmos a classe Gerente como ela está, ela vai herdar o método BuscarBonificacao.

```
Gerente gerente = new Gerente();
gerente.Salario = 5000.0;
Console.WriteLine(gerente.BuscaBonificacao());
```

O resultado aqui será 500. Não queremos essa resposta. Para consertar isso, uma das opções seria criar um novo método na classe Gerente, chamado, por exemplo, BuscaBonificacaoDoGerente. O problema é que agora teríamos dois métodos em Gerente, confundindo bastante quem for usar essa classe, além de que cada um da uma resposta diferente.

No C# temos que adicionar a palavra reservada do C# "`virtual`" na assinatura do método que será sobescrito na classe filha.

```
public class Funcionario
{
        protected String nome;
        protected String cpf;
        protected double salario;

        public virtual double BuscaBonificacao()
        {
            return this.salario * 0.10;
        }
        // métodos
}
```

No C#, quando herdamos um método, podemos alterar seu comportamento, Podemos **reescrever** (sobrescrever, override) este método:

```
class Gerente : Funcionario
{
        int senha;
        public override double BuscaBonificacao()
        {
            return this.salario * 0.15;
        }
        // ...
}
```

Agora sim, o método está correto para o Gerente. Refaça o teste e veja que agora o valor impresso é o correto (750):

```
Gerente gerente = new Gerente();
gerente.Salario = 5000.0;
Console.WriteLine(gerente.BuscaBonificacao());
```

5.3 - Invocando o método reescrito

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe: realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionário porem adicionando R$ 1000. Poderíamos fazer assim:

```
class Gerente : Funcionario
{
        int senha;
        public override double BuscarBonificacao()
        {
            return this.salario * 0.10 + 1000;
        }
        // ...
}
```

Aqui teríamos um problema: o dia que o BuscarBonificacao do Funcionario mudar, precisaremos mudar o método do Gerente para acompanhar a nova bonificação. Para evitar isso, o BuscarBonificacao do Gerente pode chamar o do Funcionario utilizando-se da palavra chave ***base***.

```
class Gerente : Funcionario
{
        int senha;
        public override double BuscarBonificacao()
        {
            return base.BuscarBonificacao() + 1000;
        }
        // ...
```

```
}
```

Essa invocação vai procurar o método com o nome BuscarBonificacao de uma super classe de Gerente. No caso ele logo vai encontrar esse método em Funcionario.

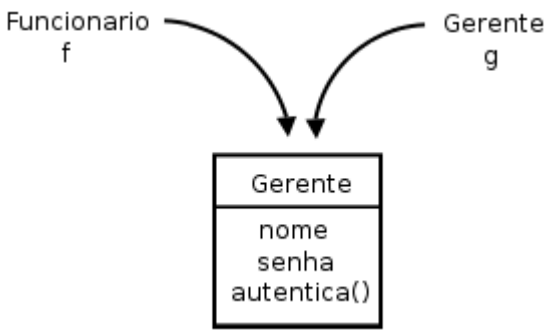
Essa é uma prática comum, pois muitos casos o método reescrito geralmente faz "algo a mais" que o método da classe mãe. Chamar ou não o método de cima é uma decisão sua e depende do seu problema. Algumas vezes não faz sentido invocar o método que reescrevemos.

5.4 - Polimorfismo

O que guarda uma variável do tipo Funcionario? Uma referência para um Funcionario.

Na herança, vimos que todo Gerente é um Funcionario, pois é uma extensão deste. Podemos nos referir a um Gerente como sendo um Funcionario. Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente! Porque? Pois Gerente **é um** Funcionario. Essa é a semântica da herança.

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.Salario = 5000.0;
```



Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.BuscaBonificacao();
```

Qual é o retorno desse método? 500 ou 750? No C#, a invocação de método sempre vai ser **decidida em tempo de execução**. O C# vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciá-lo.

Apesar de estarmos nos referenciando a esse Gerente como sendo um Funcionario, o método executado é o do Gerente. O retorno é 750.

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que faríamos isso?

Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo Funcionario:

```
class ControleDeBonificacoes
{
	private double totalDeBonificacoes = 0;

	public void registra(Funcionario funcionario)
	{
		this.totalDeBonificacoes += funcionario.BuscaBonificacao();
	}

	public double BuscaTotalDeBonificacoes()
	{
	    return this.totalDeBonificacoes;
	}
}
```

E, em algum lugar da minha aplicação (ou no main se for apenas para testes):

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();

Gerente funcionario1 = new Gerente();
funcionario1.Salario =5000.0;
controle.registra(funcionario1);

Funcionario funcionario2 = new Funcionario();
funcionario2.Salario = 1000.0;
controle.registra(funcionario2);

Console.WriteLine(controle.BuscaTotalDeBonificacoes());
```

Repare que conseguimos passar um Gerente para um método que recebe um Funcionario como argumento.

Pense como numa porta na agência bancária com o seguinte aviso: “Permitida a entrada apenas de Funcionários”. Um gerente pode passar nessa porta? Sim, pois Gerente **é um** Funcionario.

Qual será o valor resultante? Não importa que dentro do método registra do ControleDeBonificacoes receba Funcionario. Quando ele receber um objeto que realmente é um Gerente, o seu método reescrito será invocado.

Reafirmando: **não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele**.

No dia em que criarmos uma classe Secretaria, por exemplo, que é filha de Funcionario, precisaremos mudar a classe de ControleDeBonificacoes? Não. Basta a classe Secretaria reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a herança e reescrita de método: diminuir acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou ControleDeBonificacoes pode nunca ter imaginado a criação da classe Secretaria ou Engenheiro. Isso traz um reaproveitamento enorme de código.

5.5 - Um outro exemplo

Imagine que vamos modelar um sistema para a faculdade que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
class EmpregadoDaFaculdade
{
        private String nome;
        private double salario;

        public virtual double BuscarGastos()
        {
            return this.salario;
        }

        public virtual String BuscarInformacao()
        {
            return "nome: " +
             this.nome + " com salário " +
             this.salario;
        }
        // métodos de get, set e outros
}
```

O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então? Reescrevemos o método. Assim como o BuscarGastos é diferente, o BuscarInformacao também será, pois temos de mostrar as horas/aula também.

```
class ProfessorDaFaculdade : EmpregadoDaFaculdade
```

```
{
        private int horasDeAula;

        public override double BuscarGastos()
        {
            return this.Salario + this.horasDeAula * 10;
        }

        public override String BuscarInformacao()
        {
            String informacaoBasica = base.BuscaInformacao();
            String informacao = informacaoBasica +
                              " horas de aula: " + this.horasDeAula;
            return informacao;
        }
        // métodos de get, set e outros
}
```

A novidade, aqui, é a palavra chave ***base***. Apesar do método ter sido reescrito, gostaríamos de acessar o método da classe mãe, para não ter de copiar e colocar o conteúdo desse método e depois concatenar com a informação das horas de aula.

Como tiramos proveito do polimorfismo? Imagine que temos uma classe de relatório:

```
class GeradorDeRelatorio
{
        public void adiciona(EmpregadoDaFaculdade f)
        {
            Console.WriteLine(f.BuscaInformacao());
            Console.WriteLine(f.BuscaGastos());
        }
}
```

Podemos passar para nossa classe qualquer EmpregadoDaFaculdade! Vai funcionar tanto para professor, quanto para funcionário comum.

Um certo dia, muito depois de terminar essa classe de relatório, resolvemos aumentar nosso sistema, e colocar uma classe nova, que representa o Reitor. Como ele também é um EmpregadoDaFaculdade, será que vamos precisar alterar alguma coisa na nossa classe de Relatorio? Não. Essa é a idéia! Quem programou a classe GeradorDeRelatorio nunca imaginou que existiria uma classe Reitor e, mesmo assim, o sistema funciona.

```
class Reitor : ProfessorDaFaculdade
{
        // informações extras
        public override string BuscarInformacao()
        {
            return base.BuscarInformacao() + " e ele é um reitor";
        }

        // não sobreescrevemos o BuscaGastos!!!
}
```

5.6 – Exercícios: Herança e Polimorfismo

1) Vamos criar uma classe Conta, que possua um saldo, e os métodos para pegar saldo, depositar, e sacar.

a) Crie a classe Conta

```
public class Conta
{
}
```

b) Adicione o atributo saldo

```
public class Conta
{
      private string _saldo;
}
```

c) Crie a propriedade Saldo e o método Depositar(double) e Sacar(double)

```
public class Conta
{
        private double _saldo;

        public void Depositar(double valor)
        {
            this._saldo += valor;
        }

        public void Sacar(double valor)
        {
            this._saldo -= valor;
        }

        public double Saldo
        {
            get { return this._saldo; }
        }
}
```

2) Adicione um método na classe Conta, que atualiza essa conta de acordo com uma taxa percentual fornecida.

```
public class Conta
{
        private double _saldo;
        // outros métodos aqui também ...
        public void Atualizar(double taxa)
        {
            this._saldo += this._saldo * taxa;
        }
}
```

3) Crie duas subclasses da classe Conta: ContaCorrente e ContaPoupanca. Ambas terão o método atualiza reescrito: A ContaCorrente deve atualizar-se com o dobro da taxa e a ContaPoupanca deve atualizar-se com o triplo da taxa.

Além disso, a ContaCorrente deve reescrever o método deposita, afim de retirar uma taxa bancária de dez centavos de cada depósito.

Crie as classes ContaCorrente e ContaPoupanca. Ambas são filhas da classe Conta:

```
class ContaCorrente : Conta
{
}

class ContaPoupanca : Conta
{
}
```

Reescreva o método atualiza na classe ContaCorrente, seguindo o enunciado:

```
public class ContaCorrente : Conta
{
        public void Atualizar(double taxa)
        {
            this._saldo += this._saldo * taxa * 2;
        }
}
```

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

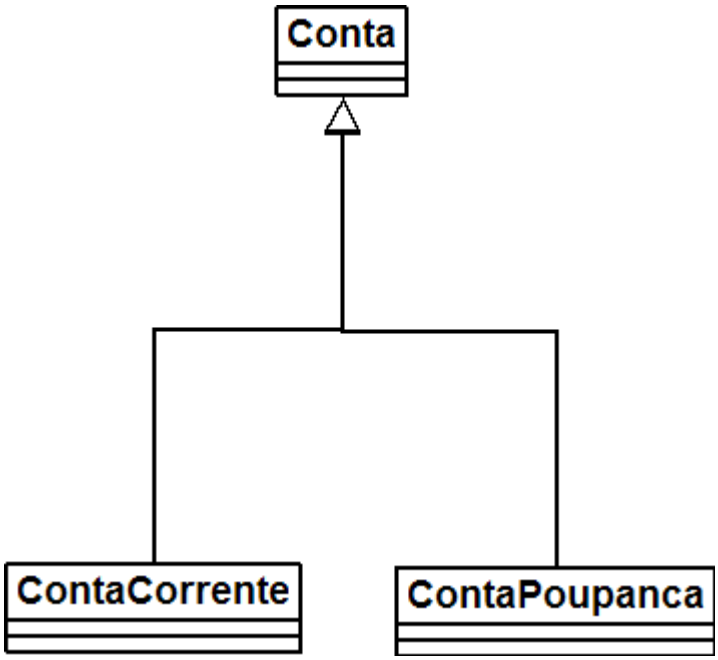
Reescreva o método atualiza na classe ContaPoupanca, seguindo o enunciado:

```
public class ContaPoupanca : Conta
{
        public void Atualizar(double taxa)
        {
            this._saldo += this._saldo * taxa * 3;
        }
}
```

Na classe ContaCorrente, reescreva o método deposita para descontar a taxa bancária de dez centavos:

```
public class ContaCorrente : Conta
{
        public void Atualizar(double taxa)
        {
            this._saldo += this._saldo * taxa * 2;
        }

        public void Depositar(double valor)
        {
            this._saldo += valor - 0.10;
        }
}
```



Observação: existem outras soluções para modificar o saldo da sua classe mãe: você pode utilizar os métodos Retirar e Depositar, se preferir continuar com o private (recomendado!), ou então criar uma propriedade Saldo, mas protected, para não deixar outras pessoas alterarem o saldo sem passar por um método (nem mesmo sua filha conseguiria burlar isso). Hoje em dia, muitas pessoas dizem que o protected quebra encapsulamento, assim como alguns casos de herança onde a mãe e filha têm um acoplamento muito forte.

4) Crie uma classe com método main e instancie essas classes, atualize-as e veja o resultado. Algo como:

```
public class TestaContas
{
        public static void main(String[] args)
        {
            Conta c = new Conta();
            ContaCorrente cc = new ContaCorrente();
            ContaPoupanca cp = new ContaPoupanca();
            c.Depositar(1000);
            cc.Depositar(1000);
            cp.Depositar(1000);
            c.Atualizar(0.01);
            cc.Atualizar(0.01);
            cp.Atualizar(0.01);
            Console.WriteLine(c.Saldo);
            Console.WriteLine(cc.Saldo);
            Console.WriteLine(cp.Saldo);
        }
}
```

Após imprimir o saldo (Saldo) de cada uma das contas, o que acontece?

5) O que você acha de rodar o código anterior da seguinte maneira:

```
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 - veremos o seu real poder no próximo exercício. Porém existe uma utilidade de declararmos uma variável de um tipo menos específico do que o objeto realmente é.

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

6) (opcional) Vamos criar uma classe que seja responsável por fazer a atualização de todas as contas bancárias e gerar um relatório com o saldo anterior e saldo novo de cada uma das contas.

```
public class AtualizadorDeContas
{
        private double _saldoTotal = 0;
        private double _selic;

        public AtualizadorDeContas(double selic)
        {
            this._selic = selic;
        }

        public void Roda(Conta c)
        {
            // aqui voce imprime o saldo anterior, atualiza a conta,
            // e depois imprime o saldo final
            // lembrando de somar o saldo final ao atributo saldoTotal
        }
        // outros métodos, colocar o getter para saldoTotal!
}
```

7) (opcional) No método main, vamos criar algumas contas e rodá-las:

```
class TestaAtualizadorDeContas
{
        public static void main(String[] args)
        {
            Conta c = new Conta();
            Conta cc = new ContaCorrente();
            Conta cp = new ContaPoupanca();
            c.Depositar(1000);
            cc.Depositar(1000);
            cp.Depositar(1000);
            AtualizadorDeContas adc = new AtualizadorDeContas(0.01);
            adc.Roda(c);
            adc.Roda(cc);
            adc.Roda(cp);
            Console.WriteLine("Saldo Total: " + adc.getSaldoTotal());
        }
}
```

8) (Opcional) Use a palavra chave ***base*** nos métodos atualiza reescritos, para não ter de refazer o trabalho.

9) (Opcional) Se você precisasse criar uma classe ContaInvestimento, e seu método atualiza fosse complicadíssimo, você precisaria alterar as classes Banco e AtualizadorDeContas?

10) (Opcional, Trabalhoso) Crie uma classe Banco que possui um array de Conta. Repare que num array de Conta você pode colocar tanto ContaCorrente quanto ContaPoupanca. Crie um método void adiciona(Conta c), um método Conta pegaConta(int x) e outro int pegaTotalDeContas(), muito similar a relação anterior de Empresa-Funcionario.

Faça com que seu método main crie diversas contas, insira-as no Banco e depois, com um for, percorra todas as contas do Banco para passá-las como argumento para o AtualizadorDeContas.