

# Unidade III

## FUNDAMENTOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

Nesta unidade, estudaremos como aplicar os conceitos fundamentais da POO, assim como a integração de tais conceitos entre si. Para auxiliar no estudo de cada conceito, utilizaremos algumas situações reais e aos poucos aplicaremos os conceitos de POO.

### Exemplo de aplicação

Não se prenda aos exemplos deste material. Tente alterá-los e procure por novos exemplos. Tente aplicá-los no seu dia a dia!

O estudante deverá ficar atento às informações como padronização de nomes e procedimentos, de modo a facilitar seu entendimento e se habituar a padrões básicos de metodologia de trabalho utilizados no mercado mundialmente.

Existe uma relação muito próxima entre os conceitos de programação estruturada e POO, assemelhando-se também em suas funcionalidades (salvo as definições que serão abordadas neste material):

**Quadro 14 – Relação de conceitos básicos da programação estruturada x POO**

Programação estruturada	Programação orientada a objetos
• Variáveis	• Variáveis • Atributos
• Procedimentos	• Métodos
• Funções	• Métodos
• Operações matemáticas e lógicas	• Operações matemáticas e lógicas
• Laços e desvio condicionais	• Laços e desvio condicionais

Conforme o quadro anterior, fica fácil perceber que os conceitos de POO se aplicam mais ao trabalho de modelagem de sistemas do que à programação em si, visto que os conceitos de programação estruturada não se perdem e são agregados com uma maneira diferente de trabalho.

### 5 CONCEITOS BÁSICOS

#### 5.1 Abstração

Abstrair consiste no ato de imaginar algo a partir de algo concreto. Em POO, abstração é o conceito mais utilizado, pois a todo momento necessitamos analisar processos e situações do mundo real e abstrai-los para a elaboração de um *software* e/ou sistema, ou seja, elencar como serão os dados e os procedimentos utilizados, tal como a interação entres esses processos.

Um exemplo é o controle de contas.

Imagine a seguinte situação: a necessidade de criar um controle financeiro para sua família.

A princípio a tarefa é simples, mas vamos abstrair o ambiente. Há alguns aspectos necessários:

- Para controlar as contas, devemos ter um lugar para anotá-las.
- Necessitamos separar as contas em contas a pagar e contas a receber.
- Normalmente, temos tipos diferentes de contas, então precisamos ter categorias (ou tipos) para agrupá-las.
- Uma vez paga ou recebida a conta, devemos controlar para onde o dinheiro foi! Logo, necessitamos de um controle de caixas.
- Caixas podem ser a carteira ou uma conta no banco.
- Depois de paga ou recebida a conta, devemos anotar que ela foi quitada.

Os processos de elencar e estudar o ambiente são realizados sempre antes de iniciar os trabalhos de codificação, ou seja, a programação em si.

O processo de abstração é necessário para definir o escopo (objetivo) de todas as classes a serem desenvolvidas, assim como o objetivo final do sistema que será construído.



#### Observação

Abstrair é imaginação! Para quem está pouco acostumado, pode ser muito difícil em um primeiro momento realizar essa análise do mundo real e traduzi-lo em desenhos simples. Somente a prática levará o estudante a ter a habilidade necessária para esse fim. Então, treinar sempre em abstrair soluções para procedimentos e eventos do dia a dia do estudante é a melhor maneira de desenvolver essa aptidão!

Em algumas linguagens (como o C#) é possível criar estruturas de classes que simplesmente não possuem nenhuma implementação concreta, apenas definições de como ela será. Isso é permitido justamente para que o desenvolvedor possa ter liberdade de abstração, ou seja, liberdade de definir seu projeto de maneira a ficar mais próximo da modelagem do mundo real.

### 5.2 Objetos

Objetos podem ser considerados uma imitação do comportamento intrínseco de entidades reais. Tal como em sistemas reais, em uma POO não é viável abrir um objeto e olhar em seu interior e tampouco alterar seu estado. Nesse paradigma, a única forma de fazer evoluir um programa é permitir que objetos compartilhem dados entre si a partir de trocas explícitas de mensagens.

Como exemplo, considere a seguinte situação:

A esposa de um jovem casal se encontra no segundo mês de gravidez. No meio da noite, a grávida acorda com uma azia terrível. Como é natural, a mulher pede que seu marido providencie um antiácido para ela.

Sob a ótica da POO, poderíamos representar a cena da seguinte maneira:

- O objeto marido recebe uma mensagem do objeto esposa.
- O objeto marido responde à mensagem da esposa mediante uma ação (buscar antiácido).
- A esposa não tem que dizer ao marido onde ele deve procurar, é responsabilidade dele procurar pelo antiácido.
- Ao objeto esposa basta ter emitido uma mensagem ao objeto marido.



#### Saiba mais

Em 2008, o Massachusetts Institute of Technology (MIT), o mesmo que desenvolveu o *sketchpad*, lançou o *scratch*. Esse programa é atualmente o padrão para o ensino da programação de computadores para crianças a partir dos oito anos. O uso desse programa facilita muito a aprendizagem da programação de computadores. Conheça o *site*:

SOBRE o *scratch*. [s.d.]. Disponível em: <[scratch.mit.edu/about/](http://scratch.mit.edu/about/)>. Acesso em: 23 out. 2014.

No *scratch*, alguns conceitos como o de mensagens podem ser utilizados. O bloco de envio de mensagem é:



Figura 33

E o de recepção é:



Figura 34

### 5.3 Classes

O desenvolvimento de classes é o resultado da abstração. Uma vez definidos os devidos escopos, teremos condições de pensar em termos de classes.

É fácil notar que muitos objetos possuem características estruturais semelhantes, embora todo objeto seja único. Um bom exemplo disso são os objetos que recebem a denominação genérica de "carro". Todos os carros possuem uma mesma estrutura, ou seja, todos os objetos carro implementam os mesmos métodos e mantêm informações sobre os mesmos atributos. O fato de que cada objeto mantém seus próprios atributos de forma encapsulada confere uma identidade única a cada objeto e também permite que dois objetos possam, eventualmente, possuir valores iguais para seus atributos.

Esses grupos de objetos com estruturas semelhantes são definidos em termos de classes. Classes consistem na unidade básica da construção de um programa OO e definem o conjunto de atributos mantidos por um conjunto de objetos e o comportamento que esses objetos devem respeitar.

#### 5.3.1 Classes em C#

O conceito de classe é genérico e pode ser aplicado em diversas linguagens de programação. Mostraremos como a classe Conta poderia ser escrita utilizando a linguagem C#. Neste momento serão apresentados apenas os atributos – os métodos serão abordados posteriormente.

## Código 18 – Primeira classe

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MinhaPrimeiraClasse
{
    class Conta
    {
        public double saldo;
        public double limite;
        public int conta;
    }

    class Program
    {
        static void Main(string[] args)
        {
            new Conta();
        }
    }
}
```

Até agora, somente trabalhamos com a classe Program, que funciona perfeitamente para a programação estruturada. Ao introduzir a POO, cada classe ocupa o mesmo nível da classe Program. No código anterior, temos, além da classe Program, a classe Conta.

A classe em C# Conta é declarada utilizando a palavra "class". No corpo dessa classe, são declarados os atributos que os objetos possuirão. Os atributos saldo e limite são do tipo *double*, que permite armazenar números com casas decimais, e o atributo numero é do tipo *int*, que permite armazenar números inteiros. O modificador *public* é adicionado em cada atributo para que eles possam ser acessados a partir de qualquer ponto do código.

### Exemplo: controle financeiro

No exemplo que apresentamos há pouco, definimos nosso escopo para gerar um controle financeiro para nossa família. Portanto, agora temos condições de iniciar um processo para estreito de abstração para definirmos quem serão nossas classes.

Analisando o escopo definido, podemos encontrar as seguintes classes:

- **Conta:** são os dados das minhas contas a pagar ou a receber.
- **Categoria:** é tipo de conta, ou seja, se são contas de telecomunicação, combustível, salário, dividendos etc.
- **Tipo de pagamento:** modalidade de pagamento, tais como dinheiro, cartão, depósito etc.
- **Caixa:** lugares onde foi usada a modalidade de pagamento. Pode ser uma carteira ou uma conta do banco.

A princípio, são essas as classes que iremos abstrair do mundo real para modelar nosso sistema em POO. O grande conceito neste ponto do nosso trabalho é: cada classe deverá ter um nome único.

Nunca poderemos ter duas classes distintas com nomes iguais. Recomenda-se, aliás, que o nome da classe seja um que condiga com seu objetivo, ou seja, evitar codificações do tipo:

- SCFF\_C001, onde: S=Sistema; C=Controle; F=financeiro; F=familiar; C=Classe; 001=Classe conta;

Organização é bom, mas nomes codificados atrapalham o processo de desenvolvimento de *software*, ainda mais se o trabalho for feito em equipe, como normalmente é.

Recomenda-se também que o nome da classe seja escrito com letras minúsculas, porém, a primeira letra deve ser maiúscula. Em caso de nomes de classes com palavras compostas, a primeira letra de cada palavra deve ser maiúscula e as demais, minúsculas. Exemplo: TipoPagamento.



### Lembrete

A POO foi criada para facilitar, para tratar o desenvolvimento de *software* de uma maneira mais orgânica e mais próxima do ser humano, deixando assim o processo de desenvolvimento de *software* mais ágil e robusto. A simplicidade é a chave!

As seguintes classes foram definidas:

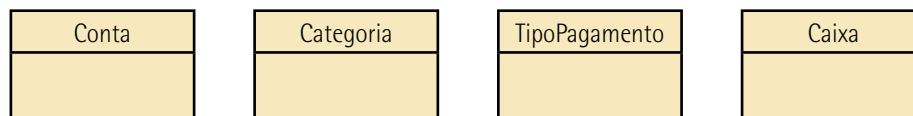


Figura 35 – Classes iniciais definidas para o controle financeiro familiar

As classes desse exemplo, traduzidas para o C#, são apresentadas no código a seguir:

## Código 19 – Apenas as classes do controle financeiro

```
namespace ProjLivro00
{
    public class TipoPagamento
    {
    }
    public class Categoria
    {
    }
    public class Caixa
    {
    }
    public class Conta
    {
    }
}
```

### 5.3.1.1 Criação de objetos em C#

O comando para criar objetos é o `new`. Veja:

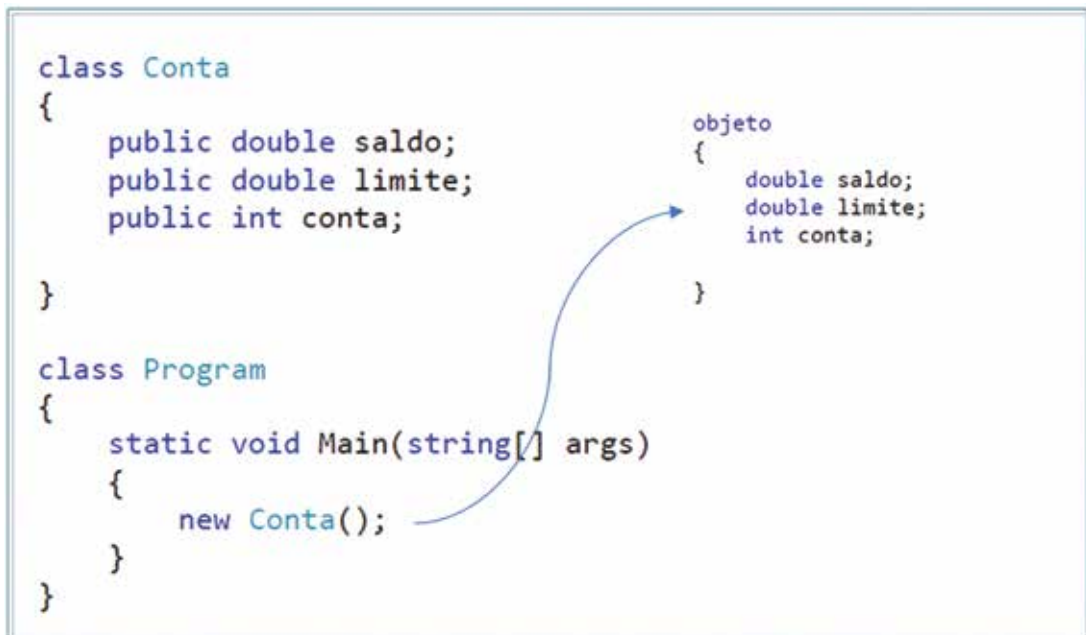


Figura 36 – Objeto criado a partir da classe Conta

Se quisermos criar três objetos na memória, utilizamos três comandos `new`:

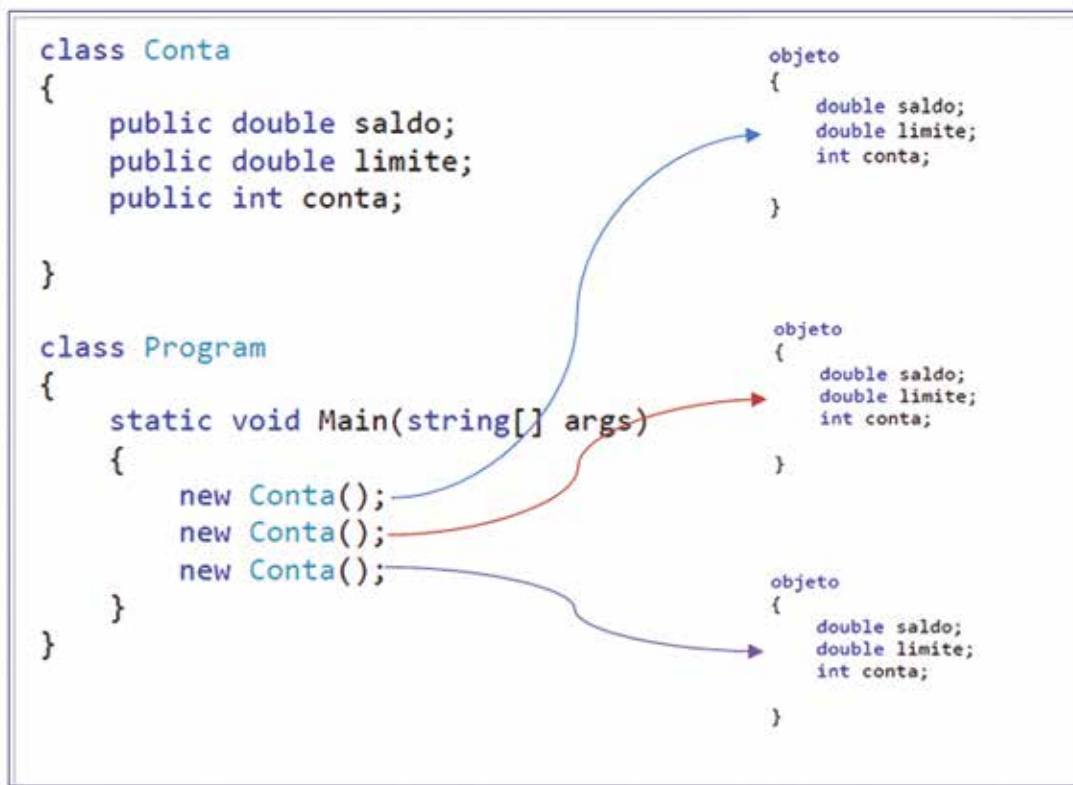


Figura 37 – Três objetos criados a partir da classe Conta

O jargão técnico para os programadores OO para o ato de criar um objeto é instanciar, ou criar uma instância. Assim sendo, um objeto também é conhecido como uma instância de uma classe.

### 5.3.1.2 Referências

Todo objeto criado possui uma referência. Ela é a única maneira de acessar os seus atributos e métodos, portanto devemos guardar as referências dos objetos que forem utilizados.

A referência é criada da seguinte forma:

```
Conta referencia = new Conta();
```

Para ser mais didático, vamos separar a linha em dois e executar passo a passo:

```
Conta referencia;
referencia=new Conta();
```

Inicialmente, uma variável é criada na memória na qual será armazenada um endereço de memória com as características de Conta:



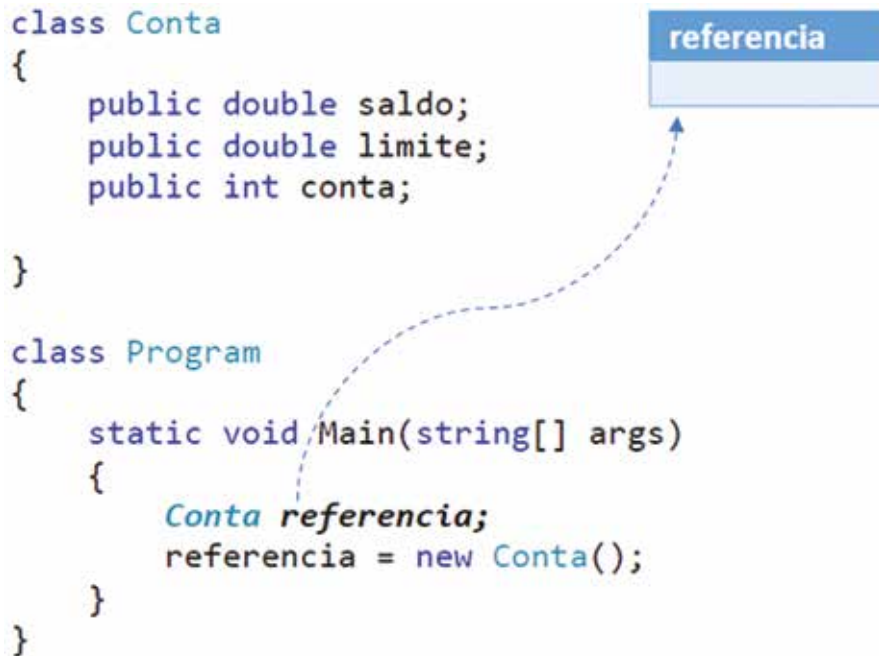


Figura 38 – É criada uma variável do tipo Conta

Como vimos anteriormente com o comando *new*, um objeto é alocado em algum lugar da memória:

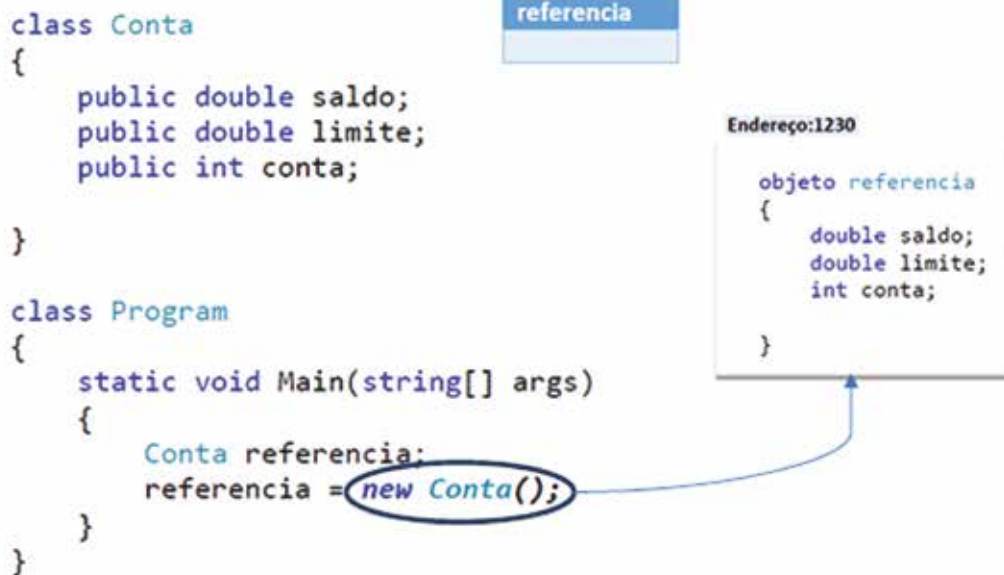


Figura 39 – O objeto é criado

Para que possamos acessar esse objeto, precisamos de sua referência. O comando *new* devolve a referência do objeto que foi criado sendo armazenado na variável criada anteriormente:

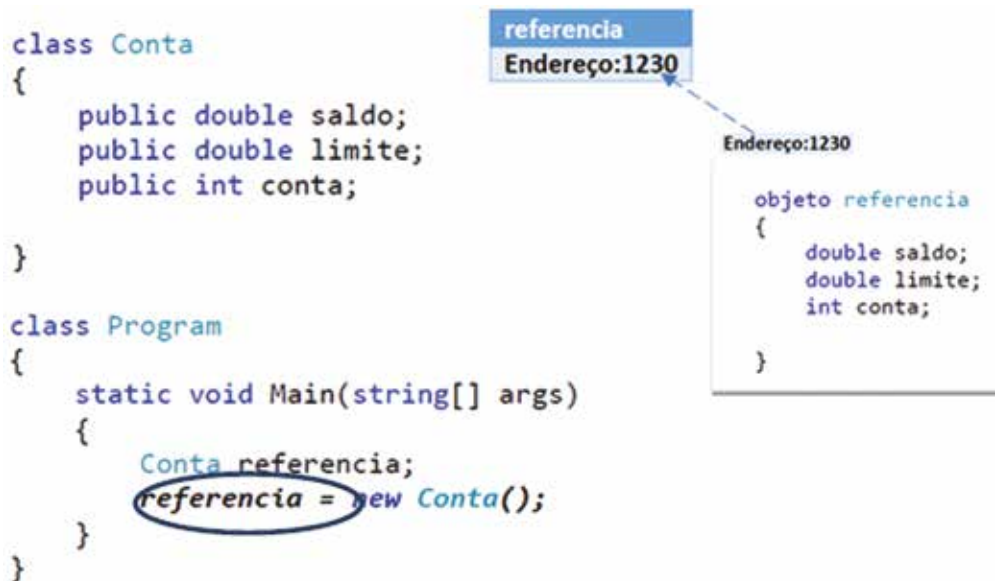


Figura 40 – Passagem da referência para a variável

A partir de então, o objeto passará a ser referenciado pelo nome dessa variável:

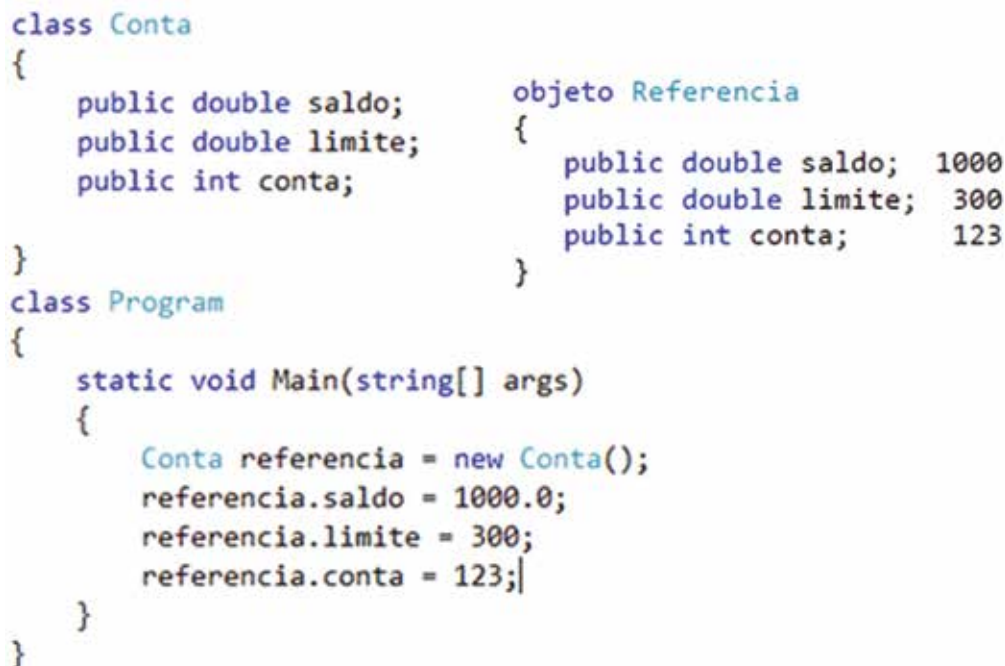


Figura 41 – Acesso ao objeto por meio da referência

Exemplo 1:

Os robôs têm como características o código de identificação, a versão do seu sistema operacional e a data de criação. A partir desse modelo, criar dois robôs, o primeiro (r2d2) com o código de identificação 1234 e o segundo (c3po) com o 9. Ambos estão com o sistema Versão V1.0 e foram criados em 991201.

Código 20 – Listagem completa do exemplo dos robôs

```
class Robo
{
    public short codigoidentificacao;
    public String versao;
    public int datacriacao;
}
class Program
{
    static void Main(string[] args)
    {
        Robo r2d2 = new Robo();
        Robo c3po = new Robo();
        r2d2.codigoidentificacao = 1234;
        r2d2.versao = "V1.0";
        c3po.versao = r2d2.versao;
        r2d2.datacriacao = c3po.datacriacao = 991201;
        c3po.codigoidentificacao = 999;
        Console.WriteLine("Robo r2d2 codigo {0} versao {1} Criacao {2}",
            r2d2.codigoidentificacao, r2d2.versao, r2d2.datacriacao);
        Console.WriteLine("Robo c3po codigo {0} versao {1} Criacao {2}",
            c3po.codigoidentificacao, c3po.versao, c3po.datacriacao);
    }
}
```

Para entendermos passo a passo o exemplo, em primeiro lugar assumiremos que os objetos receberão o nome das referências (devemos ter cuidado com isso, como veremos mais tarde). As instâncias do robô r2d2 e c3po são então criados, conforme visto na figura a seguir:

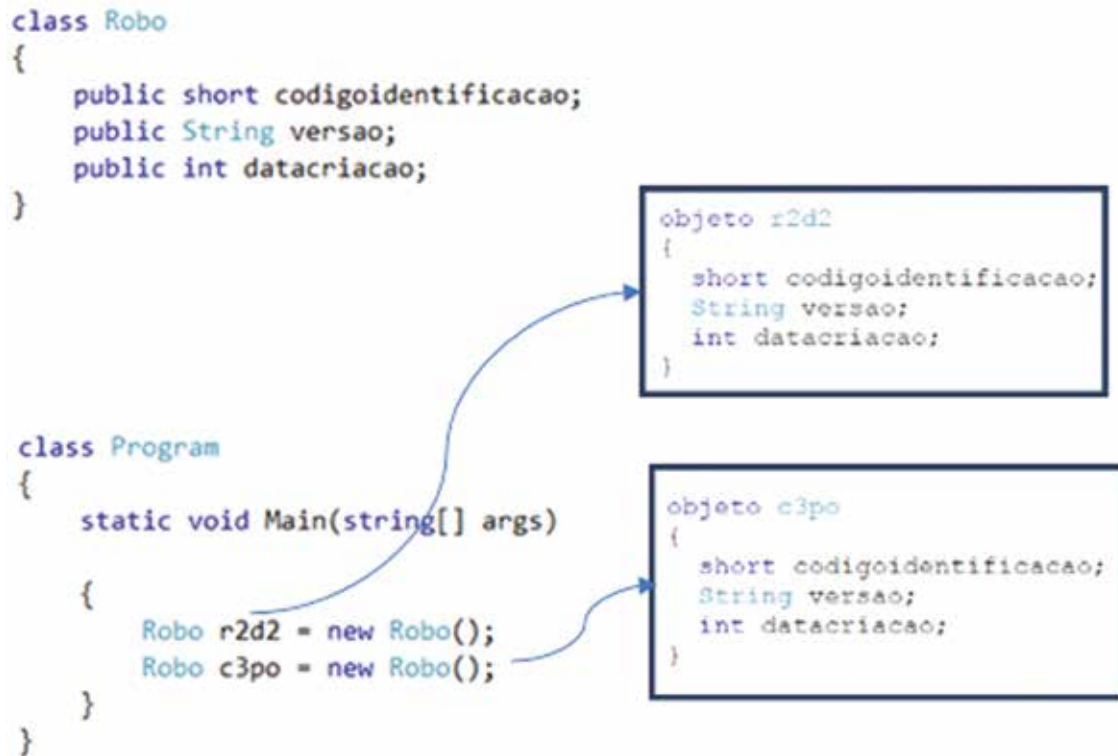


Figura 42 – Objetos r2d2 e c3po criados

Atribuímos então os valores para as instâncias utilizando as referências criadas. Assim, o código 1234 será armazenado no atributo r2d2.codigodeidentificação; o "V1.0", no atributo r2d2.versão. O atributo c3po.versão receberá o valor armazenado no atributo chamado versão que está no endereço r2d2, e ambos os atributos de data de criação receberão 991201:

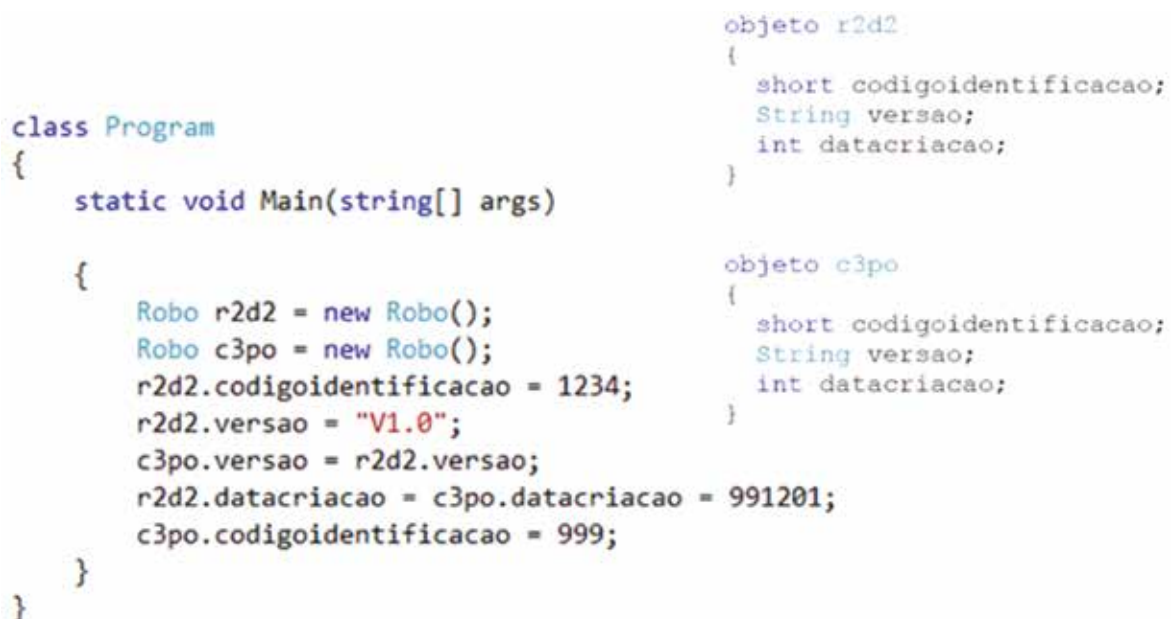


Figura 43 – Atributos dos objetos recebendo valores

A saída do programa conforme o código anterior é mostrado na figura a seguir:

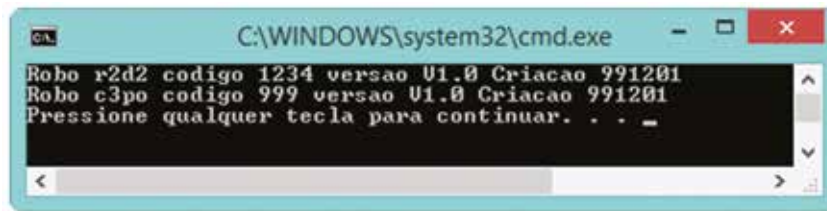


Figura 44 – Saída do programa no exemplo dos robôs

Exemplo 2:

Monte uma classe chamada Carro que guarde as seguintes características: marca, modelo, cor e ano.

Crie os seguintes carros:

Meu carro, um Volkswagen Fusca, verde, 1968, e o carro do Zé, um Gurgel BR500, branco, 1978.

## Código 21 – ExemploCarro

```
namespace ExemploCarro
{
    class Carro
    {
        public String marca;
        public String modelo;
        public String cor;
        public short ano;
    }
    class Program
    {
        static void Main(string[] args)
        {
            Carro meucarro = new Carro();
            meucarro.marca = "Volkswagen";
            meucarro.modelo = "Fusca";
            meucarro.cor = "Verde";
            meucarro.ano = 1964;
            Carro doZe = new Carro();
            doZe.marca = "Gurgel";
            doZe.modelo = "Br500";
            doZe.cor = "Branco";
            doZe.ano = 1978;
            Console.WriteLine("Meu carro é {0} modelo {1} da cor {2} ano {3}",
                meucarro.marca, meucarro.modelo, meucarro.cor, meucarro.ano);
            Console.WriteLine("O carro Zé é {0} modelo {1} da cor {2} ano {3}",
                doZe.marca, doZe.modelo, doZe.cor, doZe.ano);
        }
    }
}
```



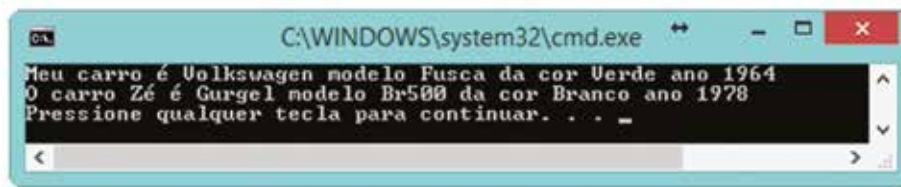


Figura 45 – Saída do ExemploCarro

Devemos ter muito cuidado com o uso de atribuição entre duas referências. É necessário lembrar que a referência é uma variável contendo um endereço de memória; assim, ao fazermos uma atribuição, **não** estamos passando o objeto de uma variável para outra, estamos passando um endereço de memória. No código a seguir acontece exatamente o problema.

### Código 22 – Passagem de endereço entre referências

```
class Gente
{
    public String nome;
}

class Program
{
    static void Main(string[] args)
    {
        Gente pessoa1 = new Gente();
        Gente pessoa2 = new Gente();
        pessoa1.nome = "Paulo";
        pessoa1 = pessoa2;
        pessoa2.nome = "Maria";
        Console.WriteLine("Pessoa1 nome={0}", pessoa1.nome);
    }
}
```

Ao executarmos, temos como saída a seguinte figura:

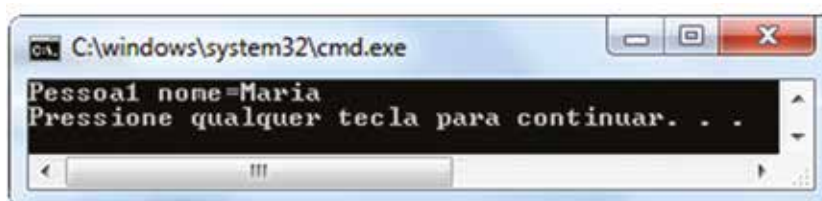


Figura 46 – Resultado da passagem de endereço

Conforme a listagem, o atributo `pessoa1.nome` recebe "Paulo" e aparentemente não mais é alterado; porém, ao pedirmos para mostrar na tela, é mostrado como "Maria"! Por que isso estaria acontecendo?

Acontece que, quando fazemos a atribuição `pessoa1=pessoa2`, ambas as variáveis passam a armazenar o mesmo endereço, e assim o primeiro objeto se perde. Tanto `pessoa1` quanto `pessoa2` passam a direcionar para o endereço do segundo objeto. Assim sendo, ao atribuirmos "Maria" ao segundo objeto, a referência `pessoa1` assume o mesmo atributo e, ao pedirmos para apresentar o valor de `pessoa1.nome`, ele apresentará "Maria".

### 5.4 Atributos

Uma vez definida a classe, devemos iniciar o processo de abstrair os atributos de cada classe, ou seja, suas características e propriedades.

Assim como na classe, os nomes dos atributos não podem se repetir dentro da classe; assim, um mesmo nome de atributo pode ser utilizado em classes diferentes, mas nunca dentro de uma mesma classe.

Recomenda-se que o nome do atributo seja um substantivo que representa a propriedade do atributo. Por padrão, também é recomendado que o nome seja escrito em minúsculo e, em caso de nomes compostos, a primeira letra a partir da segunda palavra seja escrita em maiúsculo. Exemplo: `nomeFavorecido`.



#### Observação

Em termos de POO, além dos tipos primitivos de dados existentes em cada linguagem de programação, temos o conceito de associar um atributo a uma classe. Veremos tal conceito em tópicos mais avançados deste material.

#### Exemplo: controle financeiro

Continuando o exemplo do controle financeiro visto anteriormente neste livro-texto, cada uma das classes definidas poderia apresentar os seguintes atributos:

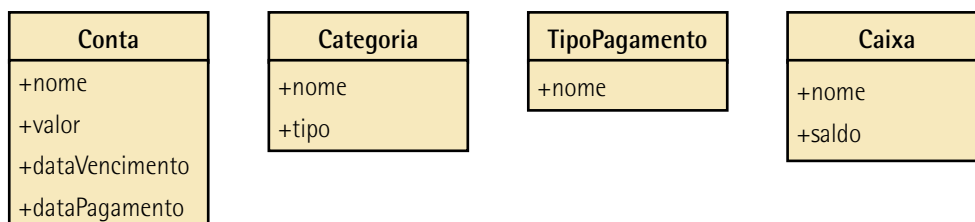


Figura 47 – Classes do controle financeiro familiar com seus atributos básicos

Percebe-se que, por exemplo, o atributo nome é utilizado em todas as classes. Porém, cada classe tem o seu atributo nome, ou seja, o mesmo atributo não está sendo compartilhado entre as classes. Existe o nome da Conta, o nome da Categoria, o nome do TipoPagamento e o nome do Caixa.

Esses atributos correspondem às características necessárias dentro do escopo de cada classe; portanto, é perfeitamente possível que algumas classes inúmeros atributos e outras classes tenham apenas um atributo. A necessidade de mais ou menos atributos em cada classe vai depender da utilização e da integração da classe em relação ao projeto.

Um grande erro que pode ocorrer neste estágio é a redundância de atributos. Redundância de atributos significa a existência da mesma informação sendo utilizada por diferentes classes de maneira não compartilhada.



### Observação

Os meios de integração entre classes no projeto serão vistos em tópicos mais avançados.



### Observação

Não podemos ter preguiça na hora de definir o nome do atributo, assim como suas quantidades. Especifique corretamente o nome, não importando o número de caracteres utilizados, e seja coerente com a necessidade de atributos na classe.

É recomendado realizar, desde o início do projeto mais simples, uma documentação sobre suas classes. Como o escopo deste material não é o gerenciamento do projeto, será utilizada aqui uma forma bem simples, porém eficiente, de documentar sua classe. Veja o quadro a seguir:

**Quadro 15 – Descritivo dos atributos em cada classe**

Classe	Atributo	Descrição
Conta	Nome	Nome da conta. Usado para descrever a conta em si.
	Valor	Valor da conta.
	dataVencimento	Data de vencimento da conta.
	dataPagamento	Data do pagamento efetivo da conta.
Categoria	Nome	Nome da categoria.
	Tipo	Tipo da categoria, pode ser crédito ou débito.
TipoPagamento	Nome	Nome da modalidade de pagamento. Exemplo: dinheiro, cheque, cartão, depósito.
Caixa	Nome	Nome do caixa.
	Saldo	Valor disponível no caixa.





## Observação

Existem várias metodologias de documentação de projeto e de documentação de código-fonte. Normalmente as equipes de desenvolvimento de *software* definem um padrão e todos os membros da equipe o seguem, para que todos "falem a mesma língua".

O exemplo de documentação dado é apenas para fins didáticos, não seguindo nenhum padrão específico.

Uma vez definidos os atributos da classe, é necessário definir também o tipo de dado utilizado em cada atributo, o que se assemelha muito à definição de tipos de dados utilizados em variáveis ou até mesmo dos campos de tabelas em bancos de dados. Dessa forma, poderíamos dizer que teríamos uma limitação técnica em relação ao tipo de linguagem de programação que seria utilizada para a implementação do *software* desenvolvido com POO. Porém, para fins didáticos, podemos utilizar neste momento o conceito de abstração para definir os tipos de dados de nossos atributos conforme o quadro a seguir:

**Quadro 16 – Tipos de dados utilizados nos atributos**

Nome	Descrição
<i>Int</i>	Tipo numérico inteiro com sinal de negativo.
<i>Double</i>	Tipo numérico real com sinal de negativo.
<i>Bool</i>	Tipo lógico. Armazena verdadeiro ( <i>true</i> ) ou falso ( <i>false</i> ).
<i>String</i>	Tipo texto. Armazena uma cadeia de caracteres.
<i>DateTime</i>	Tipo data e hora.

Logo, seguindo as especificações e necessidades de cada atributo de nossas classes, temos os seguintes diagramas:

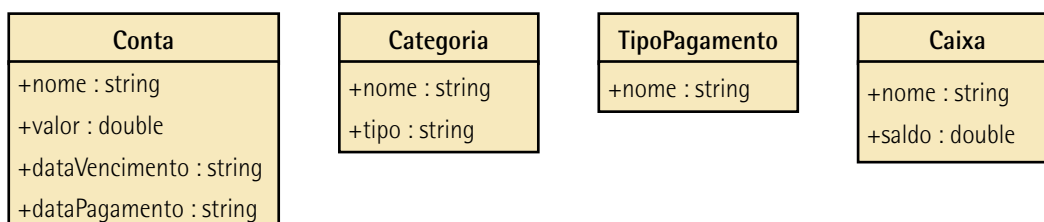
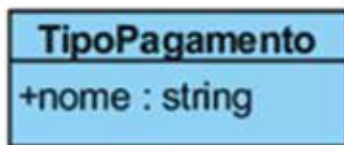
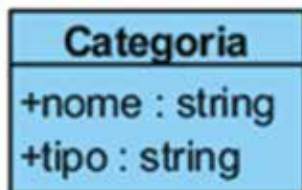


Figura 48 – Classes do controle financeiro familiar com seus atributos e respectivos tipos de dados

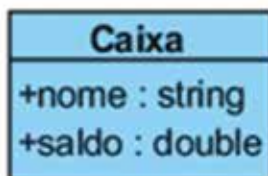
Uma vez definidos os diagramas de classe com os seus atributos, passamos a transcrevê-los em código da linguagem C#:



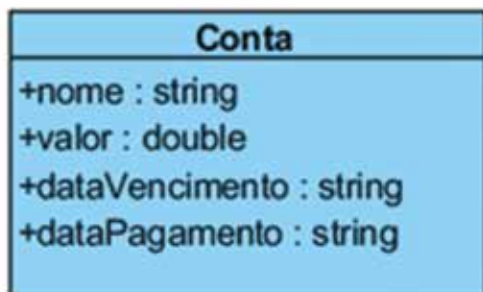
```
public class TipoPagamento
{
    public string nome;
}
```



```
public class Categoria
{
    public string nome;
    public string tipo;
}
```



```
public class Caixa
{
    public string nome;
    public double saldo;
}
```



```
public class Conta
{
    public string nome;
    public double valor;
    public string dataVencimento;
    public string dataPagamento;
}
```

Figura 49 – Passagem do diagrama de classes para o código



### Lembrete

Os tipos de dados variam entre linguagens de programação; porém, toda a linguagem de programação contempla os tipos básicos de dados para o tratamento desses, tais como dados numéricos, inteiros, lógicos e texto.

Todos os atributos de uma classe podem ser de livre acesso ou não, ou seja, os atributos podem ser públicos ou privados. Essa restrição de acesso é chamada de qualificadores de acesso. Atributos públicos são aqueles que podem ser vistos e acessados (manipulados) tanto pela classe a que pertencem como por outras. Os métodos privados são vistos e acessados (manipulados) unicamente pela classe a que pertencem.

Iremos trabalhar melhor o conceito de atributos públicos e privados posteriormente.



### Observação

Em termos de notação dos diagramas de classes, os atributos públicos são precedidos de um sinal positivo (+), e os privados, de um sinal negativo (-).

Existe um terceiro tipo de qualificador de acesso, chamado protegido. Este é simbolizado por um (#) e trataremos seu uso em momento oportuno.

## 5.5 Métodos

Dando continuidade ao trabalho de abstração, abordaremos o conceito de métodos. Um método corresponde a uma ação (operação ou comportamento) que uma classe pode realizar.

Cada método funciona de forma independente, sendo utilizado apenas quando a ação é solicitada.

Todo o método é criado seguindo uma sintaxe (regras):

- As tarefas que um objeto pode realizar são definidas pelos seus métodos.
- Os métodos são os elementos básicos para a construção dos programas OO.
- Métodos não podem ser criados dentro de outros métodos ou fora de uma classe.
- Chamada do método:
  - nomeDaInstancia.nomeDoMetodo(parâmetros).
- Todo o método é dividido em três partes:
  - nome do método;
  - argumentos (ou parâmetros);
  - retorno.

- Todo o método deve ter um nome coerente com sua função.
- Recomenda-se que o nome de cada método seja escrito com a primeira letra maiúscula e as demais, minúsculas. Em caso de nomes compostos, a primeira letra de cada palavra inicia-se maiúscula e as demais, minúsculas. Exemplo: RetornaSaldo().
- Os argumentos são declarados entre parênteses.
- Os métodos podem ter quantos argumentos forem necessários e não é obrigatório que haja argumentos.
- Os métodos podem retornar alguma resposta em relação à ação realizada, porém não é obrigatório retornar algum valor.
- Assim como nos atributos, métodos podem ser públicos ou privados.

Os métodos são, na prática, onde será realizada a programação em si, ou seja, onde a inteligência da classe será definida e onde serão executadas as funções que a classe se propõe a realizar.

Portanto, todos os conceitos da programação estruturada se aplicam na programação dos métodos das classes, ou seja, variáveis, desvios condicionais, operações matemáticas e laços (*loop*) serão utilizados.

### 5.5.1 Métodos em C#

Os métodos são definidos dentro de uma classe. Durante a execução de um programa, um método pertencente a um objeto, desde que declarados públicos podem ser acessados tanto pela referência da instância quanto pelo nome do método.

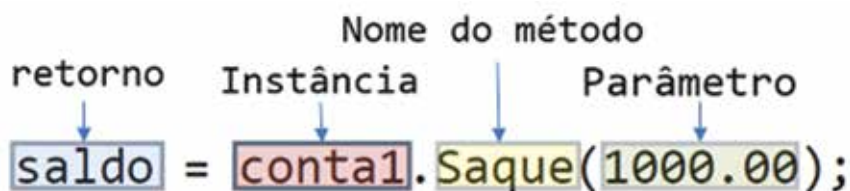


Figura 50 – Forma geral da chamada de método

Para a construção dos métodos dentro de uma classe, os seus nomes devem ser iniciados com letras (incluindo os símbolos `_` e `$`) e compostos de uma ou mais palavras sem espaço entre elas, podendo conter números. Os nomes dos métodos refletem ações que são efetuadas nos campos da classe e/ou valores passados como argumentos para esses métodos. Métodos não podem ser criados dentro de outros métodos ou fora de uma classe. Elas seguem a forma mostrada na figura a seguir:

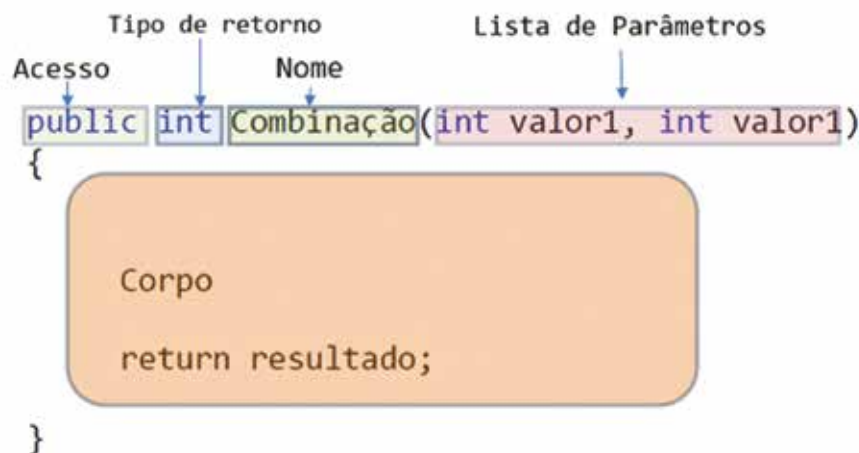


Figura 51 – Forma geral de um método

O Nome é utilizado para chamar o método. Na linguagem C#, é uma boa prática definir os nomes dos métodos utilizando a convenção "Camel Case" com a primeira letra maiúscula. A lista de parâmetros define os valores que o método deve receber. Métodos que não precisam receber nenhum valor ficam com a lista de parâmetros vazia. No corpo fica a programação estruturada que define o que acontecerá quando o método for chamado. Finalmente, o retorno é o tipo de dado que será devolvido como resposta ao final do processamento do método. Quando um método não devolve nenhuma resposta, ele deve ser marcado com a palavra reservada *void*.

Tomemos uma classe que é a conta-corrente de um cliente, com os métodos para sacar e depositar:

ContaCorrente
+Numero
+Saldo
+Limite
+Depositar()
+Sacar()

Figura 52 – Classe conta corrente com os métodos depositar e sacar

Na classe `ContaCorrente` no C#, digitamos os métodos depositar e sacar acumulando e retirando, respectivamente, do valor do saldo. Veja o código a seguir:

### Código 23 – Inserindo métodos na classe ContaCorrente

```
public class ContaCorrente
{
    public int numero;
    public String cliente;
    public float saldo;

    public void depositar(float valor)
    {
        saldo += valor;
    }
    public void sacar(float valor)
    {
        saldo -= valor;
    }
}
```

Notamos que em ambos o acesso é público e não tem retorno de valor, portanto é do tipo *void*. Ambos recebem também como parâmetro um número real que passará para dentro do método com o nome de valor.

#### 5.5.2 Métodos construtores

Os métodos construtores, ou simplesmente construtores, são métodos especiais que são chamados automaticamente quando instâncias são criadas por meio da palavra-chave *new*. Por meio da criação de construtores, podemos garantir que o código contido neles será executado antes de qualquer outro código em outros métodos.

Os construtores são úteis para iniciar campos de instâncias de classes, garantindo que, quando métodos dessas instâncias forem chamados, eles contenham valores específicos. Caso os campos de uma instância não sejam iniciados, os seguintes valores são adotados:

- Campos do tipo *boolean* são iniciados automaticamente com a constante *false*.
- Campos do tipo *char* são iniciados com o código Unicode zero, que é impresso como um espaço (ou seja: " ").
- Campos de tipos inteiros (*byte*, *short*, *int*, *long*) ou de ponto flutuante (*float*, *double*) são automaticamente iniciados com o valor zero, do tipo do campo declarado.
- Instâncias de qualquer classe, inclusive da classe *String*, são iniciadas automaticamente com *null*.

As diferenças básicas entre construtores e outros métodos são:

- A palavra-chave *null* é usada em referências, indicando que as referências não foram iniciadas mas têm algum "valor" associado a elas, apesar desse "valor" não poder ser usado como uma instância poderia.
- Construtores devem ter exatamente o mesmo nome da classe a que pertencem, inclusive considerando maiúsculas e minúsculas.
- Construtores não podem retornar nenhum valor, nem mesmo *void*, portanto devem ser declarados sem tipo de retorno;
- Construtores não devem receber modificadores como *public* ou *private* e serão públicos se a classe for pública.

A razão pela qual os construtores têm regras mais rígidas para nomenclatura que os métodos é a seguinte: quando uma instância de uma classe que tem construtores for iniciada com a palavra-chave *new*, o compilador executará automaticamente o construtor, precisando então saber exatamente qual é o nome deste. Outra diferença significativa entre construtores e métodos comuns é que o programador não pode chamar construtores diretamente – somente quando a instância for iniciada com *new*.

O construtor é um método com mesmo nome que a classe. O programador de uma classe pode definir o construtor de uma classe, que é invocado toda vez que o programa instancia um objeto desta classe (DEITEL, 2003, p. 386).

Temos a aplicação de um método construtor em uma classe no código a seguir:

## Código 24 – Método construtor

```
namespace Construtor01
{
    class Gente
    {
        public String nome;
        public int idade;
        public Gente()
        {
            nome = "José";
            idade = 20;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Gente pessoal = new Gente();
            Console.WriteLine("Nome:{0}\nidade{1}", pessoal.nome, pessoal.idade);
        }
    }
}
```

Acompanhando o processamento do programa na figura a seguir, ao ser criado o objeto `peessoa1` (1) pelo comando `new`, imediatamente o sistema passa a executar o método construtor (2).

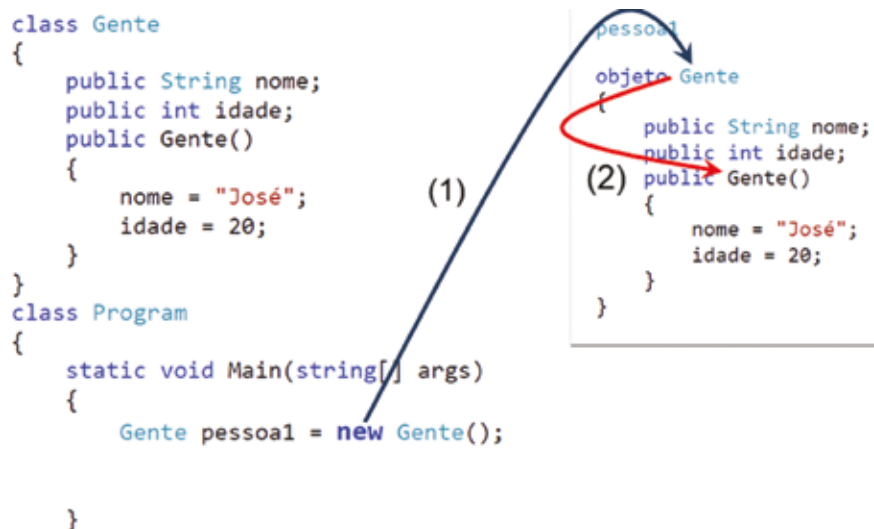


Figura 53 – Processamento do construtor

Assim ao executar esse código, mesmo sem receber os valores no programa principal, os atributos aparecem inicializados na figura a seguir:

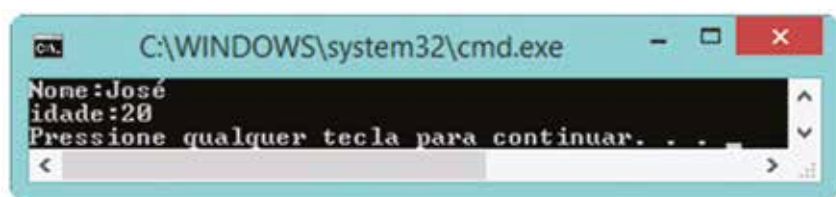


Figura 54 – Saída do programa do código anterior

### Exemplo: controle financeiro

Como já vimos a teoria, vamos agora implementar os métodos construtores padrão nas nossas quatro classes. Neste momento, o construtor só é usado para inicializar com valores neutros as classes quando é feito o processo de criação das instâncias, conforme visto nos códigos a seguir:

#### Código 25 – Implementando o método construtor na classe TipoPagamento

```
public class TipoPagamento
{
    public string nome;
    public TipoPagamento()
    {
        this.nome = "";
    }
}
```



Código 26 – Implementando o método construtor na classe Categoria

```
public class Categoria
{
    public string nome;
    public string tipo;
    public Categoria()
    {
        this.nome = "";
        this.tipo = "";
    }
}
```

Código 27 – Implementando o método construtor na classe Caixa

```
public class Caixa
{
    public string nome;
    public double saldo;
    public Caixa()
    {
        this.nome = "";
        this.saldo = 0;
    }
}
```

Código 28 – Implementando o método construtor na classe Conta

```
public class Conta
{
    public string nome;
    public double valor;
    public string dataVencimento;
    public string dataPagamento;
    public Conta()
    {
        this.nome = "";
        this.valor = 0;
        this.dataVencimento = "";
        this.dataPagamento = "";
    }
}
```

## 5.5.3 Sobrecarga de métodos

Conforme a necessidade, é útil ou interessante ser possível executar um método em uma classe passando mais ou menos argumentos. Linguagens orientadas a objeto permitem a criação de métodos com nomes iguais, contanto que as suas assinaturas sejam diferentes. Na programação estruturada, é

inimaginável ter dois ou mais funções com o mesmo nome. A assinatura de um método é composta de seu nome mais os tipos de argumentos que são passados para esse método, independentemente dos nomes de variáveis usadas na declaração do método.

O tipo de retorno não é considerado parte da assinatura; não podemos ter dois métodos com o mesmo nome e tipo de argumentos mas tipo de retorno diferente.

A possibilidade de criar mais de um método com o mesmo nome e assinaturas diferentes é conhecida como sobrecarga de métodos. A decisão sobre qual método será chamado quando existem dois ou mais métodos será feita pelo compilador, baseado na assinatura dos métodos (SANTOS, 2003, p. 77).

A sobrecarga é válida para qualquer tipo de método, mesmo o construtor. Assim sendo, no código a seguir temos a sobrecarga do método construtor da classe *Gente*. Como vemos, cada um dos métodos com mesmos nomes possuem a composição de tipos diferentes como parâmetros em cada um deles.

### Código 29 – Sobrecarga do método construtor

```
class Gente
{
    public String nome;
    public int idade;
    public Gente()
    {
        nome = "José";
        idade = 20;
    }
    public Gente(String valor)
    {
        nome = valor;
        idade = 20;
    }
    public Gente(int valor)
    {
        nome = "José";
        idade = valor;
    }
    public Gente(String valorn, int valori)
    {
        nome = valorn;
        idade = valori;
    }
}
```

Veja as figuras a seguir. O primeiro construtor é o que não recebe nenhum parâmetro; o segundo recebe como parâmetro uma *string*, portanto a assinatura fica *Gente* e *string*; o terceiro recebe um inteiro; e o quarto recebe na sequência *string* e *int*.

```
public Gente()  
{  
    nome = "José";  
    idade = 20;  
}
```

assinatura Gente()

Figura 55 – Assinatura vazia

```
public Gente(String valor)  
{  
    nome = valor;  
    idade = 20;  
}
```

assinatura Gente(String)

Figura 56 – Assinatura *String*

```
public Gente(int valor)  
{  
    nome = "José";  
    idade = valor;  
}
```

assinatura Gente(int)

Figura 57 – Assinatura *int*

```
public Gente(String valorn, int valori)  
{  
    nome = valorn;  
    idade = valori;  
}
```

assinatura Gente(String int)

Figura 58 – Assinatura *string int*



## Observação

A assinatura obedece à sequência dos tipos. Assim sendo, se no exemplo anterior tivéssemos mais um método cujos parâmetros fossem respectivamente *int* e *string*, ele teria uma assinatura diferente do *string* e

*int* presentes no programa. Portanto, a ordem dos tipos é considerada na assinatura.

No exemplo do método construtor, ao incluirmos a classe a seguir no momento em que a nova instância é criada, o sistema procura o construtor que possui a mesma assinatura e o executa (veja a figura a seguir).

### Código 30 – Chamada de um método sobrecarregado

```
class Program
{
    static void Main(string[] args)
    {
        Gente pessoa1 = new Gente("Maria");
        Console.WriteLine("Nome:{0}\nidade:{1}",
            pessoa1.nome, pessoa1.idade);
    }
}
```

```
public Gente(String valor)
{
    nome = valor;
    idade = 20;
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Gente pessoa1 = new Gente("Maria");
        Console.WriteLine("Nome:{0}\nidade:{1} ",
            pessoa1.nome, pessoa1.idade);
    }
}
```

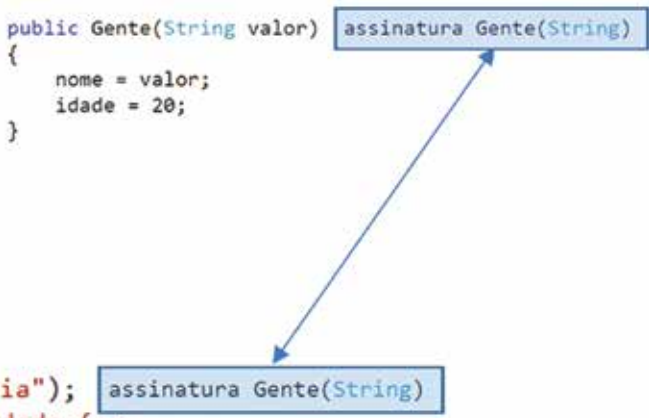


Figura 59 – Chamada de método sobrecarregado

Assim sendo, o atributo nome assume o valor passado como parâmetro:

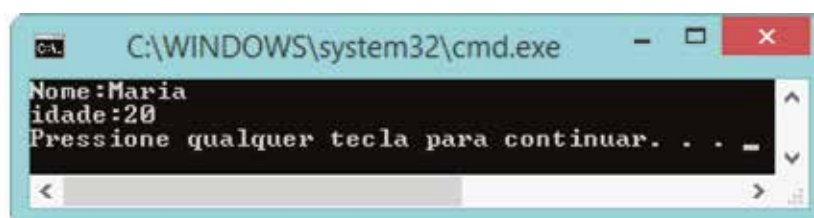


Figura 60 – Saída do código anterior

Outro exemplo envolve métodos que não são construtores. É o que temos no código a seguir.

## Código 31 – Sobrecarga de métodos

```
class Gerente
{
    public String nome;
    public double salario;
    public void AumentaSalario()
    {
        this.AumentaSalario(0.1);
    }
    public void AumentaSalario(double taxa)
    {
        this.salario += this.salario * taxa;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Gerente g = new Gerente();
        g.salario = 1000;
        Console.WriteLine("Salário: " + g.salario);
        Console.WriteLine("Aumentando o salário em 10% ");
        g.AumentaSalario();
        Console.WriteLine("Salário: " + g.salario);
        Console.WriteLine("Aumentando o salário em 30% ");
        g.AumentaSalario(0.3);
        Console.WriteLine("Salário: " + g.salario);
    }
}
```

O exemplo mostra o caso de um reajuste de salário, em que por padrão se aumenta 10%; porém, é possível também reajustar a partir de uma taxa qualquer. Como temos dois métodos com o mesmo nome `AumentaSalario`, porém com assinaturas diferentes, o sistema automaticamente procura o método que tem a assinatura correspondente e o executa. A primeira chamada é feita com a lista de parâmetros vazia, assim o sistema procura o método com a assinatura vazia, e aplica a taxa de 10%. Na segunda chamada, é passado como parâmetro um valor com casas decimais, e então o sistema procura o método que comporta esse tipo de assinatura, no caso *double*, e faz o cálculo com o índice passado na chamada. Veja:

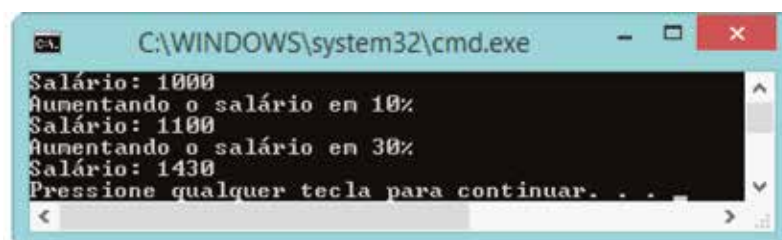


Figura 61 – Saída do código anterior

### 5.5.4 A palavra-chave *this*

Um método pode chamar outro método de uma mesma classe. Para os construtores, a tarefa é mais complicada (não se pode chamar um construtor diretamente). Então, é criada internamente para cada instância uma "auto referência", ou seja, uma referência à própria instância. Essa referência é representada pela palavra-chave *this*. Para chamar um construtor de dentro do outro, basta usar a palavra-chave *this* substituindo o nome do construtor. Construtores não podem ser chamados indiscriminadamente de dentro de qualquer método; existem algumas regras para a chamada de construtores, conforme se verá a seguir:

- Somente construtores podem chamar construtores como sub-rotinas.
- Se um construtor for chamado a partir de outro, a chamada deve ser a primeira linha de código dentro do corpo do construtor.
- Construtores não são chamados pelos seus nomes, e sim por *this*.
- Construtores podem chamar outros métodos. Por exemplo, pode ser interessante ter um construtor e um método que iniciem as instâncias e chamar o método de dentro do construtor. Métodos não podem chamar construtores, nem mesmo com *this*.
- Construtores não podem ser chamados recursivamente. Um construtor só pode chamar diretamente outro construtor e não a si próprio.

O uso mais imediato é como uma autorreferência dos atributos dentro do objeto (veja o código a seguir). Nele, a palavra-chave passa a se referir diretamente aos atributos da classe.

#### Código 32 – Autorreferência *this*

```
class Gente
{
    public String nome;
    public int idade;
    public Gente(String nome, int idade)
    {
        this.nome = nome;
        this.idade = idade;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Gente pessoa1 = new Gente("Paulo",23);
    }
}
```

Poderíamos fazer uma analogia com o coringa do baralho. A palavra *this* passa a substituir a referência, conforme visto na figura a seguir. Dessa forma, o atributo nome passa a ser outra entidade, diferente do parâmetro nome, que é passado pela criação do objeto.

```
class Gente
{
    public String nome;
    public int idade;
    public Gente(String nome, int
    {
        this.nome = nome;
        this.idade = idade;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Gente pessoa1 = new Gente("Paulo",23);
    }
}
```

The diagram illustrates the execution of the constructor. On the left, the `Gente` class constructor is shown with two lines: `this.nome = nome;` and `this.idade = idade;`. On the right, the creation of an object is shown: `Gente pessoa1 = new Gente("Paulo",23);`. Below this, the state of the object is shown: `pessoa1.nome = nome;` and `pessoa1.idade = idade;`. Two curved arrows originate from the `this` keyword in the constructor and point to the `pessoa1` object, indicating that `this` refers to the instance being created.

Figura 62 – *This* especificando um atributo

Um construtor tem o mesmo nome da classe, mas esta, ao se tornar um objeto, passa a ter o nome do objeto. Assim sendo, não é possível que a codificação chame explicitamente um método construtor pelo nome da classe. Como vimos, a palavra-chave *this* atua como um coringa desta, que assume o nome da instância. Portanto, em vez de chamar um construtor pelo nome da classe, basta chamá-lo pela palavra *this*.

A linguagem C#, ao contrário das outras linguagens OO, necessita de uma construção especial, menos intuitiva. Assim, quando se quer uma autorreferência nos construtores, ela deve ser colocada após a declaração do método, separada por dois pontos. Podemos então alterar o programa exemplo da sobrecarga de métodos, o que faz com que os diversos construtores chamem o método mais completo, evitando a codificação individual de cada um deles. Todos chamam o método cuja assinatura é *string* e nome:

### Código 33 – Alterando o código da sobrecarga de métodos para utilizar o *this*

```
class Gente
{
    public String nome;
    public int idade;
    public Gente()
        : this("José", 20)
    {
    }
    public Gente(String nome)
        : this(nome, 20)
    {
    }
    public Gente(int valor)
        : this("José", valor)
    {
    }
    public Gente(String nome, int idade)
    {
        this.nome = nome;
        this.idade = idade;
    }
}
```



#### Observação

Um cuidado adicional deve ser tomado quando se criam métodos sobrecarregados: normalmente, os programas OO permitem que alguns tipos nativos de dados sejam promovidos, isto é, aceitos como sendo de outros tipos, contanto que nada se perca na representação. Dessa forma, um valor do tipo *byte* pode ser aceito por um método que espere um valor do tipo *int*, já que este pode representar *bytes* sem perda de informação.

#### Exemplo: controle financeiro

Já estávamos utilizando a palavra-chave *this* no nosso código, porém sem a explicação técnica. Podemos ver que ela se autorreferencia dentro da própria instância quando criada. Para podermos ter mais opções de inicialização no momento em que é o objeto é criado, é feita a sobrecarga dos construtores nas classes do nosso sistema. Observe os códigos a seguir:



Código 34 – Uso do *this* e sobrecarga do construtor na classe TipoPagamento

```
public class TipoPagamento
{
    public string nome;
    public TipoPagamento()
    {
        this.nome = "";
    }
    public TipoPagamento(string fnome)
    {
        this.nome = fnome;
    }
}
```

Código 35 – Uso do *this* e sobrecarga do construtor na classe Categoria

```
public class Categoria
{
    public string nome;
    public string tipo;
    public Categoria()
    {
        this.nome = "";
        this.tipo = "";
    }
    public Categoria(string fnome, string ftipo)
    {
        this.nome = fnome;
        this.tipo = ftipo;
    }
}
```

Código 36 – Uso do *this* e sobrecarga do construtor na classe Caixa

```
public class Caixa
{
    public string nome;
    public double saldo;
    public Caixa()
    {
        this.nome = "";
        this.saldo = 0;
    }
    public Caixa(string fnome, double fsaldo)
    {
        this.nome = fnome;
        this.saldo = fsaldo;
    }
}
```

Código 37 – Uso do *this* e sobrecarga do construtor na classe Conta

```
public class Conta
{
    public string nome;
    public double valor;
    public string dataVencimento;
    public string dataPagamento;

    public Conta()
    {
        this.nome = "";
        this.valor = 0;
        this.dataVencimento = "";
        this.dataPagamento = "";
    }
    public Conta(string fnome, double fvalor)
    {
        this.nome = fnome;
        this.valor = fvalor;
    }
}
```

Vejamos a aplicação dos métodos no exemplo do sistema de controle financeiro:

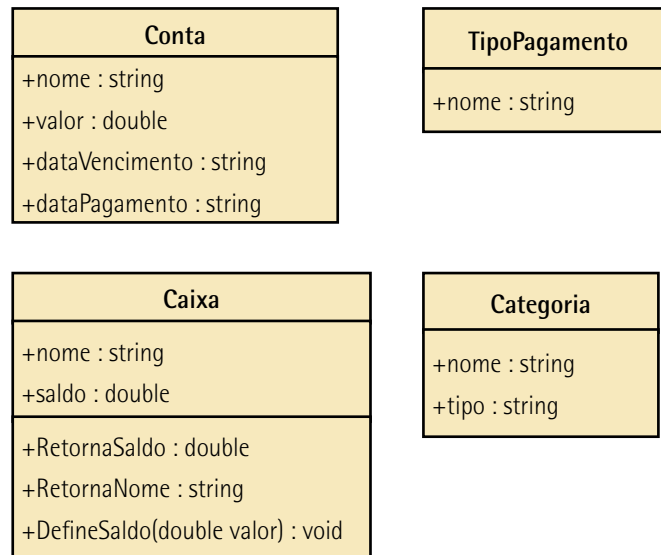


Figura 63 – Classes do controle financeiro familiar com a classe Caixa e seus métodos

De acordo com a figura, temos a classe Caixa com seus respectivos métodos. Cada método corresponde a uma operação específica que a classe pode realizar, conforme descrito no quadro a seguir:

## Quadro 17 – Descrição dos métodos da classe Caixa

Nome	Parâmetros	Retorno	Descrição
RetornaSaldo	Nenhum	O saldo do caixa.	Retorna o valor atual do caixa
RetornaNome	Nenhum	O nome do caixa.	Retorna o nome do caixa
DefineSaldo	Valor a ser armazenado no caixa	O saldo do caixa.	Retorna o saldo atual do caixa

Assim, alteramos o código da classe Caixa para incluir os métodos recém-modelados.

### Código 38 – Inserindo os métodos na classe Conta

```
public class Conta
{
    public string nome;
    public double valor;
    public string dataVencimento;
    public string dataPagamento;

    public Conta()
    {
        this.nome = "";
        this.valor = 0;
        this.dataVencimento = "";
        this.dataPagamento = "";
    }
    public Conta(string fnome, double fvalor)
    {
        this.nome = fnome;
        this.valor = fvalor;
    }

    public double RetornaSaldo()
    {
        return this.valor;
    }
    public String RetornaNome()
    {
        return this.nome;
    }
    public void DefineSaldo(double valor)
    {
        this.valor = valor;
    }
}
```

Percebe-se que os métodos da classe Caixa seguiram as regras de como construir métodos. Porém, se analisarmos toda a estrutura da classe Caixa, percebemos que:

- A classe possui métodos e atributos.
- Dois métodos (RetornaSaldo e RetornaNome) retornam o valor de dois atributos (saldo e nome, respectivamente).

- O método DefineSaldo manipula o valor do atributo saldo.

Desta forma, percebemos que quando alguém for utilizar essa classe em outra parte do programa, tanto faz acessar e manipular os atributos ou os métodos! Isso pode gerar uma série de problemas e inconsistências que podem comprometer toda a modelagem a realizar e, por consequência, a eficácia da programação realizada.

Para evitar esse tipo de problema, alguns cuidados devem ser tomados. Isso pode ser feito adotando algumas regras adicionais na modelagem da classe que visam proteger e guiar o uso da classe de forma intuitiva, padronizada e transparente:

- Limite o acesso ao máximo dos atributos pelas demais classes (atributos privados).
- Padronize a manipulação e o retorno de valores dos atributos utilizando os métodos.

Vamos ver no próximo item pormenorizadamente como efetuar a limitação do acesso aos atributos.



## Observação

As regras apresentadas neste material referente ao nome de classes e atributos, assim como as estruturas utilizando qualificadores de acesso, são fundamentadas em boas práticas de programação, nada tendo a ver com regras de sintaxe de linguagem de programação. Tais práticas podem e devem ser adotadas pela equipe de desenvolvimento de *software* e podem ser diferentes em vários ambientes (escolas, empresas, departamentos etc.).

## 6 INTEGRAÇÃO ENTRE CLASSES

Conforme discutido anteriormente, uma boa prática é ocultar a estrutura de dados de uma classe, de modo a padronizar os acessos dos dados da classe por meio de métodos.

A seguir, serão apresentados alguns conceitos que permitem integrar classes de maneira a construir um sistema coeso, relacionando as classes existentes.

### 6.1 Encapsulamento

Antes de entrarmos nos detalhes, vamos estudar um caso: abstrair uma conta corrente simples (veja a figura a seguir). Simplificando ao máximo, temos um nome e o valor do dinheiro depositado (saldo). Essa classe tem dois métodos, uma para depósito e outra para saque, e as movimentações acontecem por meio deles.

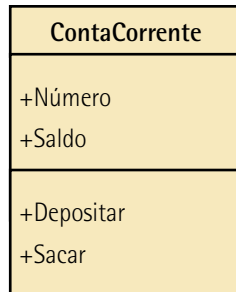


Figura 64 – Classe conta corrente simples

A codificação dessa classe fica:

### Código 39 – Classe Conta

```
class Conta
{
    public String nome;
    public double saldo;
    public Conta(String nome)
    {
        this.nome = nome;
        this.saldo = 0;
    }
    public void depositar(double valor)
    {
        this.saldo += valor;
    }
    public void sacar(double valor)
    {
        this.saldo -= valor;
    }
}
```

Acrescentamos a classe Program, na qual faremos alterações.

## Código 40 – Classe Program inicial

```
class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta("José Frederico");
        Console.WriteLine("Nome:{0}\nSaldo:{1}",
            ccJF.nome, ccJF.saldo);
    }
}
```

A saída inicial mostra a conta inicializada com o saldo zero:

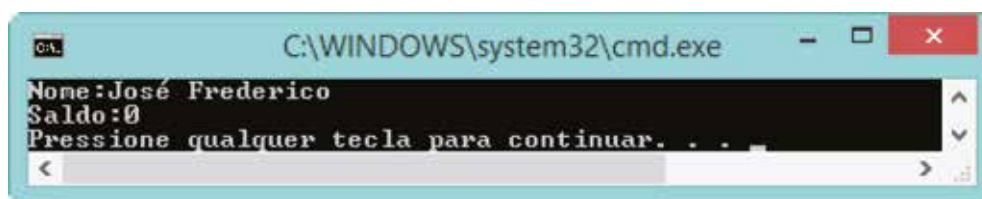


Figura 65 – Saída inicial do programa

Continuando, com o programa acrescentaremos um depósito de R\$ 500,00:

## Código 41 – Depósito de R\$ 500,00

```
class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta("José Frederico");
        ccJF.depositar(500);
        Console.WriteLine("Nome:{0}\nSaldo:{1}",
            ccJF.nome, ccJF.saldo);
    }
}
```

Com isso, o valor do saldo fica atualizado:

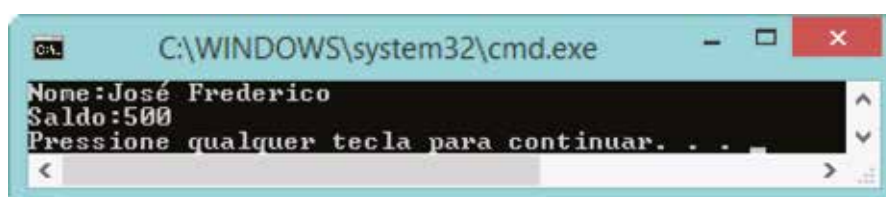


Figura 66 – Saldo após o depósito

Agora, vamos acessar diretamente o atributo saldo do objeto ccJF e mudar para R\$1.000.000,00.

### Código 42 – Mudança no saldo

```
class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta("José Frederico");
        ccJF.depositar(500);
        ccJF.saldo = 1000000;
        Console.WriteLine("Nome:{0}\nSaldo:{1}",
            ccJF.nome, ccJF.saldo);
    }
}
```

Com isso, o saldo passa a ter o valor novo, sendo desprezado o depósito anterior:

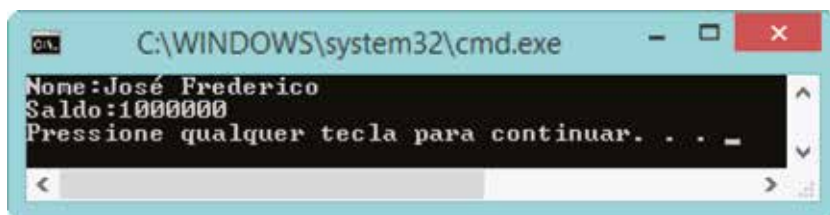


Figura 67 – Saldo após a alteração direta

Já que existe saldo disponível, é possível fazer um saque de R\$200.000,00, restando ainda com um saldo para utilizar:

### Código 43 – Saque de R\$ 200.000,00

```
class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta("José Frederico");
        ccJF.depositar(500);
        ccJF.saldo = 1000000;
        ccJF.sacar(200000);
        Console.WriteLine("Nome:{0}\nSaldo:{1}",
            ccJF.nome, ccJF.saldo);
    }
}
```



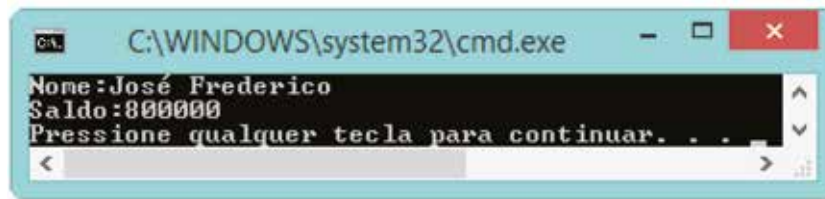


Figura 68 – Saldo após o saque

Acontece então um problema sério: considerando a movimentação bancária simples, entre depósitos e saques, houve uma perda de confiabilidade, pois no total foram depositados R\$ 500,00 e sacados R\$200.000,00. O valor do saldo pode ser alterado por fora do objeto, permitindo acontecer o erro. Para evitar esses problemas de acessos indevidos é que se faz o encapsulamento.

O conceito de encapsulamento diz respeito ao uso de classes, a partir de seus métodos públicos, por outras classes. Esse conceito nos diz como interligar classes onde existam dependências. Ele é o mecanismo utilizado para disponibilizar métodos, protegendo o acesso direto indevido aos atributos de uma instância (objeto). O encapsulamento evita a interferência externa indevida de um objeto sobre os dados de outros objetos a ele referenciados.

## 6.1.1 Modificadores de acesso

Uma das principais vantagens do paradigma da orientação a objetos é a possibilidade de encapsular os campos, bem como os métodos capazes de manipular esses campos em uma classe. É desejável que os campos das classes fiquem ocultos ou escondidos dos programadores usuários dessas classes para evitar que os dados sejam manipulados diretamente, mas que sejam manipulados apenas por intermédio dos métodos da classe. A restrição ao acesso a atributos e métodos em classes é feita por meio de modificadores de acesso que são declarados dentro das classes, antes dos métodos e dos campos. A restrição de acesso é estabelecida na definição da classe usando um dos modificadores: *private*, *protected*, *public* ou sem informação.

- **Modificador *public*:** garante que o atributo ou método da classe declarado com esse modificador possa ser acessado ou executado a partir de qualquer outra classe, ou seja, sem restrição. Os atributos e métodos que devam ser acessados (e modificados, no caso de campos) precisam ser declarados com ele.
- **Modificador *private*:** atributos ou métodos declarados com esse modificador só podem ser acessados, modificados ou executados por métodos da própria classe, sendo completamente ocultos para o programador usuário que usar instâncias desta classe ou criar classes herdeiras ou derivadas. Campos ou métodos que devam ser completamente ocultos de usuários da classe precisam ser declarados com este modificador.
- **Modificador *protected*:** funciona como o modificador *private*, exceto pela diferença de que classes herdeiras ou derivadas também terão acesso ao campo ou método marcado com esse modificador. Assim, é permitido o acesso a todas as classes derivadas.

As regras básicas para implementação de políticas para classes simples são:

- Todos os campos de uma classe devem ser declarados com o modificador *private* ou *protected*, ficando assim ocultos para o programador usuário dessas classes.
- Métodos que devam ser acessíveis devem ser declarados explicitamente com o modificador *public*.
- Como a princípio os campos terão o modificador *private*, métodos que permitam a manipulação controlada dos valores dos campos (conhecidos por métodos "getters" e "setters" ou encapsulamento) devem ser escritos nas classes e ter o modificador *public*.
- Na medida do necessário, os métodos de uma classe podem ser declarados com o modificador *private* – esses métodos não poderão ser executados por classes escritas por programadores usuários, mas poderão ser executados por outros métodos dentro da mesma classe.

Com esses conceitos, podemos voltar ao problema do acesso direto ao saldo da classe Conta.

Para evitar que o saldo não possa ser modificado externamente, vamos trocar o modificador do atributo saldo para *private*:

### Código 44 – Alteração no modificador do atributo saldo

```
class Conta
{
    public String nome;
    private double saldo;
```

O simples fato de trocar o modificador escondeu o atributo saldo da classe Conta. Desse modo, ele ficou inacessível externamente, como demonstra a figura a seguir. Porém, a informação do saldo é necessária externamente. O fato de tirar o acesso direto ao atributo passou a limitar a visibilidade do seu conteúdo.

```
class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta("José Frederico");
        ccJF.depositar(500);
        ccJF.saldo = 1000000;
        ccJF.sacar(200000);
        Console.WriteLine("Nome:{0}\nSaldo:{1}",
            ccJF.nome, ccJF.saldo);
    }
}
```

double Conta.saldo

Error:

'NecessidadeEncapsular.Conta.saldo' is inaccessible due to its protection level

Figura 69 – Mensagem de erro provocada pela alteração do modificador

Para tornar visível externamente, criamos um método (getSaldo()) para acessar a leitura apenas do conteúdo do atributo saldo:

### Código 45 – Método getSaldo()

```
class Conta
{
    public String nome;
    private double saldo;
    public Conta(String nome)
    {
        this.nome = nome;
        this.saldo = 0;
    }
    public void depositar(double valor)
    {
        this.saldo += valor;
    }
    public void sacar(double valor)
    {
        this.saldo -= valor;
    }
    public double getSaldo()
    {
        return this.saldo;
    }
}
```

Dessa forma, o programa externo também necessita ser alterado. Agora o valor do atributo fica disponível apenas para a leitura passando pelo método getSaldo():

### Código 46 – Alteração no acesso externo para visualizar o valor do atributo saldo

```
class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta("José Frederico");
        ccJF.depositar(500);
        // ccJF.saldo = 1000000;
        ccJF.sacar(200000);
        Console.WriteLine("Nome:{0}\nSaldo:{1}",
            ccJF.nome, ccJF.getSaldo());
    }
}
```

Assim foi feito o encapsulamento do atributo saldo; porém, conforme as regras básicas para implementação de políticas para classes simples, é recomendável que todos os atributos fiquem *private* ou *protected*, assim temos mais um atributo que está visível para as outras classes, o nome. Como o atributo nome poderá ser alterado externamente, mas sem acesso direto, monta-se um método para inserir um valor ao atributo, conforme visto no código a seguir. Para essa técnica, dá-se o nome de "getters" e "setters".

### Código 47 – Get e set do Nome

```
private String nome;
public String getNome()
{
    return this.nome;
}
public void setNome(String nome)
{
    this.nome = nome;
}
```

O acesso a partir de outras classes é feito por meio dos métodos `setNome()`, para atribuir um valor a nome, e `getNome()`, para receber o valor contido no atributo nome:

## Código 48 – Trabalhando com *getters* e *setters*

```
class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta("");
        ccJF.setNome("Jose Frederico");
        Console.WriteLine("Nome:{0}",ccJF.getNome());
    }
}
```

O nome `getNome` e `setNome` é apenas uma convenção. O *get* indica que um valor será devolvido e o *set* indica que um valor será dado em um atributo chamado `nome`.

Temos no quadro a seguir um resumo completo do processo de encapsulamento de um atributo público.

**Quadro 18 – Refatoração (encapsulamento) de um atributo público**

Sem encapsular	Encapsulamento
<code>public Tipo atributo;</code>	<pre>private Tipo atributo; public Tipo getAtributo() {     return this.atributo; } public void setAtributo(Tipo valor) {     this.atributo = valor; }</pre>
<code>Referencia.atributo=x;</code>	<code>Referencia.setAtributo(x);</code>
<code>x=Referencia.atributo;</code>	<code>x=Referencia.getAtributo();</code>

A forma geral aceita em todas as linguagens OO de encapsulamento é dos métodos dos "*getters*" e "*setters*", porém a linguagem C# permite outras formas de encapsulamento. Uma delas é encapsular sem transformar em método, mas criando uma variável de passagem:

### Código 49 – Encapsulamento no C#

```
class Conta
{
    private String nome;

    public String Nome
    {
        get { return nome; }
        set { nome = value; }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Conta ccJF = new Conta();
        ccJF.Nome = "Jose Frederico";
        Console.WriteLine("Nome:{0}", ccJF.Nome);
    }
}
```

Dessa maneira, o atributo nome fica protegido por uma variável de passagem Nome. No caso do saldo, em que é somente permitida a leitura, o encapsulamento fica conforme se verá no código a seguir. Sem dúvida, economiza-se algumas linhas, mas perde-se a vantagem de acessar os atributos por meio dos métodos:

### Código 50 – Encapsulamento apenas para a leitura

```
private double saldo;

public double Saldo
{
    get { return saldo; }
}
```

As classes foram atualizadas para que todos os atributos se tornassem privados (acessados apenas pela classe a que pertencem) e com seus métodos públicos.

Dessa maneira, as classes ganharam uma padronização:

- Os valores dos atributos apenas serão alterados caso se utilize o método correspondente.
- Os valores dos atributos serão acessados apenas pelos métodos correspondentes.

Classes construídas dessa forma garantem robustez e transparência no desenvolvimento do sistema. Posteriormente, veremos com mais detalhes a aplicação dessa vantagem.

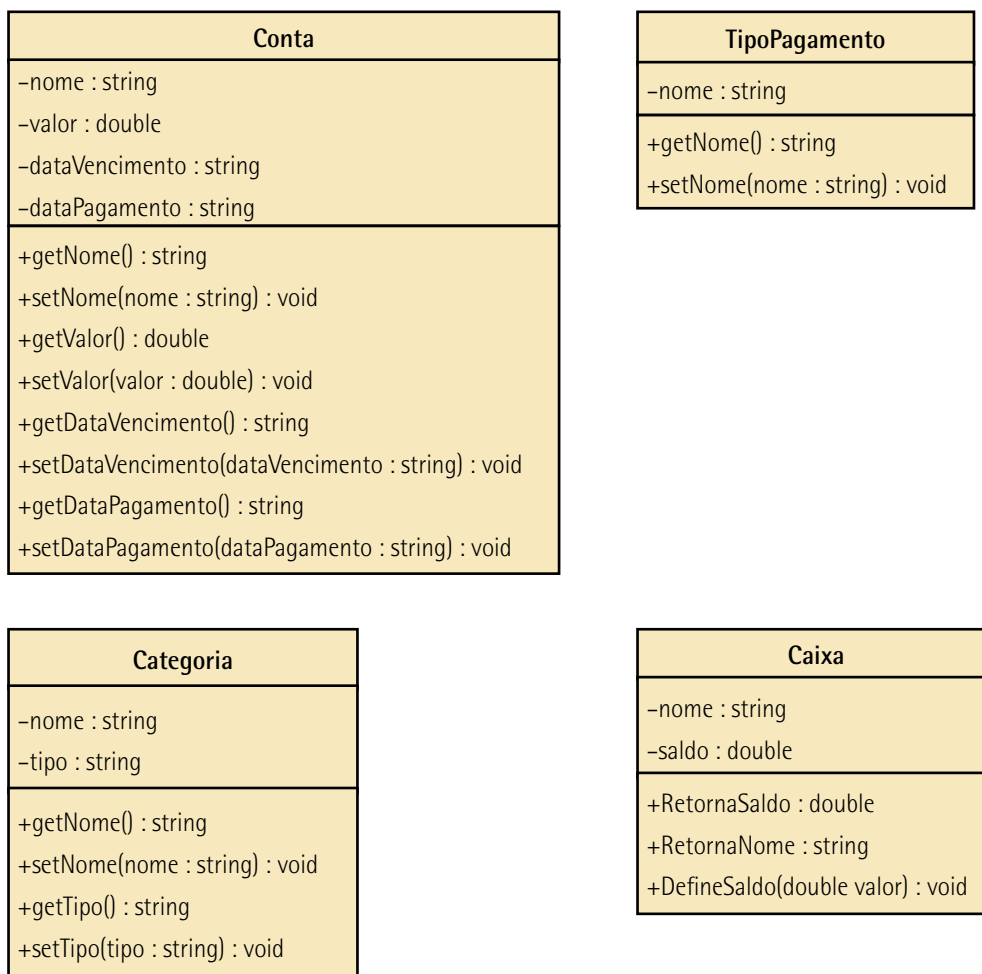


Figura 70 – Classes do controle financeiro familiar com a classe Caixa, seus métodos e seus respectivos qualificadores de acesso: público (+) e privado (-)

As classes TipoPagamento, Categoria e Caixa (vistas nos códigos a seguir) são bem simples. O encapsulamento foi realizado utilizando a técnica dos *getters* e *setters* do atributo nome, deixando assim a classe mais simples e organizada.

### Código 51 – Encapsulando a classe TipoPagamento

```
public class TipoPagamento
{
    private String nome;
    public String getNome()
    {
        return nome;
    }
    public void setNome(String nome)
    {
        this.nome = nome;
    }
}
```

### Código 52 – Encapsulando a classe Categoria

```
public class Categoria
{
    private String nome;
    private String tipo;
    public String getNome()
    {
        return nome;
    }
    public void setNome(String nome)
    {
        this.nome = nome;
    }
    public String getTipo()
    {
        return tipo;
    }
    public void setTipo(String tipo)
    {
        this.tipo = tipo;
    }
}
```



## Código 53 – Encapsulando a classe Caixa

```
public class Caixa
{
    private String nome;
    private double saldo;

    public String getNome()
    {
        return nome;
    }
    public void setNome(String nome)
    {
        this.nome = nome;
    }
    public double getSaldo()
    {
        return saldo;
    }
    public void setSaldo(double saldo)
    {
        this.saldo = saldo;
    }
}
```

Na classe Conta, conforme se verá no próximo código, também houve o encapsulamento, porém o nome dos métodos foi mantido como na modelagem original propositalmente. Como os nomes dos métodos não são caracterizados pela presença das palavras *get* ou *set*, à primeira vista tem-se a impressão de que não está sendo efetuado o encapsulamento. Com a prática, isso pode trazer problemas, já que o uso dos *getters* e *setters* é difundido e, sem o uso de uma padronização, pode induzir o programador a uma interpretação incorreta do programa.

Nessa classe também foi usado o encapsulamento especializado da linguagem C# no atributo nome, que externamente é visível como Nome (com N maiúsculo).

### Código 54 – Encapsulamento além de utilizar as palavras *get* e *set* na classe Conta

```
public class Conta
{
    private String nome;
    public string Nome
    {
        set
        {
            nome = value;
        }
        get
        {
            return nome;
        }
    }
    private double valor;
    public String dataVencimento;
    public String dataPagamento;
    public Conta()
    {
        this.nome = "";
        this.valor = 0;
        this.dataVencimento = "";
        this.dataPagamento = "";
    }

    public Conta(string fnome, double fvalor)
    {
        // Comandos de iniciação
        this.nome = fnome;
        this.valor = fvalor;
    }

    public double RetornaSaldo()
    {
        return this.valor;
    }

    public String RetornaNome()
    {
        return this.nome;
    }

    public double DefineSaldo(double valor)
    {
        this.valor = valor;
        return this.RetornaSaldo();
    }
}
```

Assim como na classe `TipoPagamento`, os métodos de definição e recuperação de dados das propriedades da classe `Conta` foram substituídos pelos `set` e `get`, respectivamente. Entretanto, as propriedades `categoriaConta`, `tipoPagamentoConta` e `caixaConta`, como são heranças de outras classes (conforme já vimos neste material), neste exemplo, são acessadas diretamente pelo usuário da classe, pois os demais controles estão dentro de cada uma das respectivas classes.

Neste item, vimos na prática o primeiro pilar da POO, o encapsulamento.

### 6.1.2 Modificador *static*

Os modificadores *static* também são conhecidos como campos estáticos em classes. Os campos estáticos de uma classe são compartilhados por todas as instâncias dela, isto é, somente um valor será armazenado em um campo estático e, caso esse valor seja modificado por uma das instâncias da classe, a modificação será refletida em todas as outras instâncias.

Campos estáticos são declarados com o modificador *static*, que deve ser declarado antes do tipo de dado do campo e pode ser combinado com modificadores de acesso como *public* e *private*. Campos estáticos também são conhecidos como campos de classes, já que esses campos poderão ser acessados diretamente usando o nome da classe, sem que haja a necessidade da criação de uma instância da classe e de uma referência para tal instância.

A palavra *static* possui as seguintes características:

- É utilizada nas declarações de constantes globais e em valores que necessitam ser manipulados em todas as classes.
- Pode ser usada com classes, campos, métodos, propriedades, operadores, eventos e construtores, mas não com indexadores, destrutores ou tipos diferentes de classes.
- Uma classe estática não pode ser instanciada.
- Uma mensagem pode ser direcionada diretamente a uma classe por meio de uma invocação a um método estático.

A utilização do modificador fica clara quando utilizamos um exemplo. No código a seguir, temos um programa que acrescenta um ao atributo `seq` no construtor da classe `Exemplo`:

## Código 55 – Programa sem o modificador *static*

```
public class Exemplo
{
    private int seq = 0;
    public int nro;
    public Exemplo()
    {
        nro = ++seq;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Exemplo e1 = new Exemplo();
        Console.WriteLine(e1.nro);

        Exemplo e2 = new Exemplo();
        Console.WriteLine(e2.nro);

        Exemplo e3 = new Exemplo();
        Console.WriteLine(e3.nro);
    }
}
```

Desse modo, a cada instância criada, o atributo seq inicia com o valor 0; assim, cada uma das instâncias mostram o valor 1 contido no atributo nro:

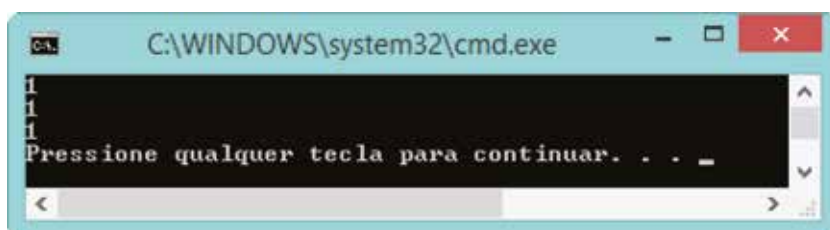


Figura 71 – Saída sem o modificador

Vamos alterar a classe Exemplo acrescentando o modificador *static* no atributo seq e executar o mesmo programa:

Código 56 – Acréscimo do modificador *static* no atributo seq

```
public class Exemplo
{
    private static int seq = 0;
    public int nro;
    public Exemplo()
    {
        nro = ++seq;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Exemplo e1 = new Exemplo();
        Console.WriteLine(e1.nro);

        Exemplo e2 = new Exemplo();
        Console.WriteLine(e2.nro);

        Exemplo e3 = new Exemplo();
        Console.WriteLine(e3.nro);
    }
}
```

Ao executar o programa, vemos que o atributo seq não é mais zerado a cada nova instância criada. Quando a primeira instância é criada, o atributo seq também é; porém, ele será o mesmo para todas as instâncias criadas. O atributo ficará estático na memória; assim, a cada nova instância, a contagem prossegue, mantendo o valor relativo à instância anterior:

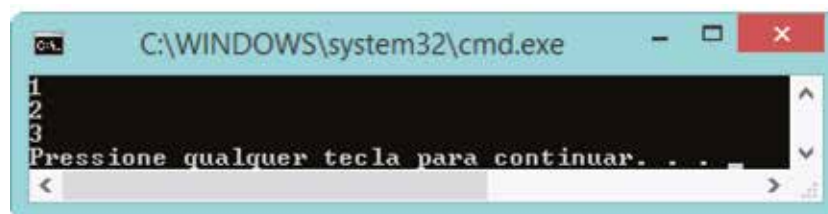


Figura 72 – Resultado com o modificador *static*

### 6.2 Associações

Uma associação representa um relacionamento entre classes e fornece a semântica comum e a estrutura para muitos tipos de "conexões" entre objetos.

Em uma associação, determina-se que as instâncias de uma classe estão de alguma forma ligadas às instâncias das outras classes envolvidas na associação, podendo haver troca de informações entre elas e compartilhamento de métodos, mesmo que determinada instância de uma das classes origine uma ou mais instâncias nas outras classes envolvidas na associação.

As associações são representadas por retas ligando as classes envolvidas, podendo também possuir setas em suas extremidades para indicar a navegabilidade da associação, o que representa o sentido em que as informações são transmitidas entre as classes envolvidas. Quando não houver setas, significa que as informações podem trafegar entre todas as classes da associação.

Além disso, as associações também podem possuir títulos para determinar o tipo de vínculo estabelecido entre as classes. Embora não seja obrigatório definir uma descrição para a associação, é útil determinar um nome para ela quando é necessária alguma forma de esclarecimento.

As associações representam o equivalente mais próximo dos relacionamentos utilizados no modelo entidade-relacionamento, ou seja, seu objetivo é definir a maneira como as classes estão unidas e se relacionam entre si, compartilhando informações (GUEDES, 2006, p. 72).

### 6.3 Agregação e composição

Agregação é um tipo especial de associação em que se tenta demonstrar que as informações de um objeto (chamado objeto-todo) precisam ser complementadas pelas informações contidas em um ou mais objetos de outra classe (chamado objeto-parte). Ela define uma dependência fraca entre as classes, ou seja, os objetos continuam existindo mesmo que o todo seja removido. Assim, se temos um time de basquete e pessoas como jogadores, ambos existirão independentemente.

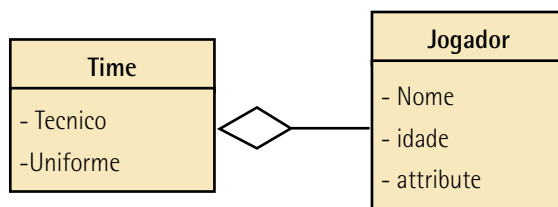


Figura 73 – Exemplo de agregação

Constitui-se numa variação da associação de agregação. A composição tenta representar um vínculo mais forte entre os objetos-todo e os objetos-parte, procurando demonstrar que os objetos-parte têm de pertencer exclusivamente a um único objeto-todo com que se relacionam.

Em uma composição, um mesmo objeto-parte não pode se associar a mais de um objeto-todo. Assim, se o todo deixar de existir, as partes deixam de existir e, se as partes deixarem de existir, o todo

desaparece. O exemplo mais típico é o do pedido de compras. Ele não existirá se não houver itens, assim como itens não fazem sentido se não estiverem agrupados em um pedido.

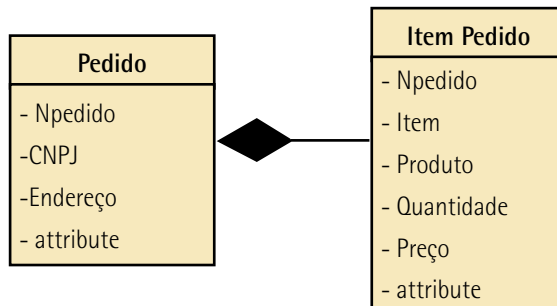


Figura 74 – Exemplo de composição

Como exemplo de associação, podemos criar uma classe chamada **Motor**, na qual temos como característica a quantidade de cilindros e o volume do motor em litros, e outra classe chamada **Carro**, cujas características são a cor e o motor que o equipa.

## Código 57 – Classe Motor

```
class Motor
{
    public int Cilindros;
    public double Capacidade;
    public Motor(int cil, double cc)
    {
        this.Cilindros = cil;
        this.Capacidade = cc;
    }
}
```

## Código 58 – Classe Carro

```
class Carro
{
    public string cor;
    public Motor motor;
    public Carro(String cor, Motor motor)
    {
        this.cor = cor;
        this.motor = motor;
    }
}
```

Assim, para criarmos três carros, um Gol 1000 vermelho, um Fox 1.0 prata e um CrossFox 1.6 amarelo,



Figura 75 – Instâncias de carro com diferentes motorizações

teremos a seguinte montagem:

### Código 59 – Montagem dos três Carros

```
class Carro
class Program
{
    static void Main(string[] args)
    {
        Motor AP1000 = new Motor(4, 1.0);
        Motor AP1600 = new Motor(4, 1.6);
        Carro gol1000 = new Carro("Vermelho", AP1000);
        Carro fox1000 = new Carro("Preto", AP1000);
        Carro crossFox = new Carro("Amarelo", AP1600);
        Console.WriteLine(crossFox.motor.Capacidade);
    }
}
```

Inicialmente, montamos os motores AP1000 e AP1600; depois, quando é feita a instância dos carros, os motores são colocados nos objetos. Note que a informação da capacidade do motor é feita utilizando duas referências, crossFox e motor.

## 6.4 Reutilização de classes

Uma das características mais interessantes de linguagens POO é a capacidade de facilitar a reutilização de código – o aproveitamento de classes e seus métodos que já estejam escritos e que já tenham o seu funcionamento testado e comprovado. A reutilização de código diminui a necessidade de escrever novos métodos e classes, economizando o trabalho do programador e diminuindo a possibilidade de erros.



Existem dois mecanismos de reaproveitamento de classes em Java: delegação e herança. Com delegação, usamos uma instância de classe base como campo na nova classe e, com herança, criamos a nova classe como uma extensão direta da classe base.

### 6.4.1 Delegação

O primeiro mecanismo de reaproveitamento de classes é conhecido como delegação (ou composição, o mesmo visto em associações). Podemos criar novas classes que estendem outra classe base se incluirmos uma instância da classe base como um dos campos da nova, que será então composta de campos específicos e de uma instância de uma classe base. Para que os métodos da classe base possam ser executados, escreveremos métodos correspondentes na classe nova que chamam os da classe base, delegando dessa forma a execução dos métodos.

Podemos criar novas classes que estendem outra classe base se incluirmos uma instância da classe base como um dos campos da nova classe, que será então composta de campos específicos e de uma instância de uma classe base. Para que os métodos da classe base possam ser executados, escreveremos métodos correspondentes na classe nova que chamam os da classe base, desta forma delegando a execução dos métodos (SANTOS, 2003, p. 174).

Exemplo: considere a classe Data, que representa uma data, e a classe Hora, que representa uma hora. Ambas contêm métodos que iniciam e verificam a validade de seus campos e imprimem os valores de seus campos em um formato apropriado. Com essas classes podemos criar a classe DataHora, que representa simultaneamente uma data e uma hora, sem que seja necessário reescrever os campos e métodos contidos nas classes Data e Hora.

#### Código 60 – Classe Data

```
class Data
{
    int dia; int mês; int ano;
    public Data(int d, int m, int a)
    {
        dia = d; mês = m; ano = a;
    }
    public override String ToString()
    { return dia + "/" + mês + "/" + ano; }
}
```

### Código 61 – Classe Hora

```
public class Hora
{
    int hora; int minuto; int segundo;

    public Hora(int h, int m, int s)
    {
        hora = h; minuto = m; segundo = s;
    }
    public override String ToString()
    { return hora + ":" + minuto + ":" + segundo; }
}
```

### Código 62 – Classe DataHora

```
class DataHora
{
    private Data estaData; private Hora estaHora;
    public DataHora(int hora, int minuto, int segundo, int dia, int mês, int ano)
    {
        estaData = new Data(dia, mês, ano);
        estaHora = new Hora(hora, minuto, segundo);
    }
    public DataHora(int dia, int mês, int ano)
    {
        estaData = new Data(dia, mês, ano);
        estaHora = new Hora(0, 0, 0);
    }
    public override String ToString()
    { return estaData.ToString() + " " + estaHora.ToString(); }
}
```

Os únicos campos na classe `DataHora` são uma instância da classe `Data` e uma instância da classe `Hora`. Todas as informações que devem ser representadas por uma classe `DataHora` estarão contidas nessas instâncias. Campos adicionais poderiam ser declarados, se fosse necessário.

O primeiro construtor da classe `DataHora` recebe seis argumentos, correspondentes ao dia, mês, ano, hora, minuto e segundo que devem ser representados pela classe `DataHora`, e repassa esses argumentos para os construtores que iniciarão as instâncias das classes `Data` e `Hora`. O construtor da classe `DataHora` delega aos outros construtores a iniciação dos campos. O mesmo acontece com o segundo construtor da classe `DataHora`, com a diferença que este considera que a hora é, por padrão, meia-noite.

O método `ToString` também delega o seu funcionamento aos métodos `ToString` das classes `Data` e `Hora`, que retornarão *strings* que são concatenadas para criar o resultado da chamada do método `ToString` da classe `DataHora`.



## Observação

A vantagem da reutilização, neste exemplo, é o fato de a nova classe `DataHora` ser capaz de representar simultaneamente uma data e uma hora sem ser muito complexa. A complexidade (capacidade de verificar se a data ou hora estão corretas etc.) é implementada pelos métodos das classes `Data` e `Hora`, que são simplesmente reutilizados.

Para completar o exemplo vamos implementar a classe `Program` para ver o mecanismo de delegação funcionando:

### Código 63 – Classe Program

```
class Program
{
    static void Main(string[] args)
    {
        DataHora agora = new DataHora(10, 5, 30, 12, 10, 2014);
        Console.WriteLine(agora.ToString());
    }
}
```

Ao executarmos, o programa apresenta como saída o que se vê na figura a seguir. Os valores 10, 5, 30, 12, 10 e 2014 são passados para o construtor da classe `DataHora`, que por sua vez monta a data 12/10/2014 e hora 10:05:30. Isso é apresentado pelo método `ToString()`.

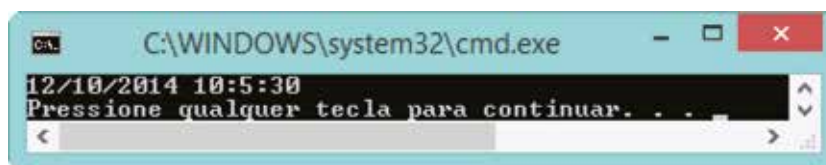


Figura 76 – Saída do programa de delegação `DataHora`

### Exemplo: controle financeiro

Para demonstrar o efeito do encapsulamento, o código a seguir mostra três composições criadas na classe `Conta`: `categoriaConta`, `tipoPagamentoConta` e `caixaConta`.

Quando um atributo é definido como uma classe, no nosso caso, o atributo `categoriaConta` é do tipo `Categoria`, um exemplo de composição. Dizemos então que a classe `Categoria` está encapsulada dentro da classe `Conta` a partir do atributo `categoriaConta`.

### Código 64 – Composição das classes Categorias, TipoPagamento e Caixa na classe Conta

```
public class Conta
{
    private String nome;
    public string Nome
    {
        set
        {
            nome = value;
        }
        get
        {
            return nome;
        }
    }

    private double valor;
    public String dataVencimento;
    public String dataPagamento;
    public Categoria categoriaConta;
    public TipoPagamento tipoPagamentoConta;
    public Caixa caixaConta;

    public Conta()
    {
        this.nome = "";
        this.valor = 0;
        this.dataVencimento = "";
        this.dataPagamento = "";
        categoriaConta = new Categoria();
        tipoPagamentoConta = new TipoPagamento();
        caixaConta = new Caixa();
    }

    public Conta(string fnome, double fvalor)
    {
        // Comandos de iniciação
        this.nome = fnome;
        this.valor = fvalor;
        categoriaConta = new Categoria();
        tipoPagamentoConta = new TipoPagamento();
        caixaConta = new Caixa();
    }

    public double RetornaSaldo()
    {
        return this.valor;
    }
    public String RetornaNome()
    {
        return this.nome;
    }
    public double DefineSaldo(double valor)
    {
        this.valor = valor;
        return this.RetornaSaldo();
    }
}
```



## Observação

Os atributos criados por composição de classes no exemplo do código anterior foram definidos como atributos públicos meramente para fins didáticos. A definição de um atributo ser ou não público vai depender unicamente do objetivo da classe e da modelagem de dados realizada pelo desenvolvedor do sistema. Tecnicamente, nada impede de definir esses atributos como privados e criar métodos específicos para acessá-los.

Estamos prontos para testar o programa. Os atributos `categoriaConta`, `tipoPagamentoConta` e `caixaConta`, neste exemplo, por serem delegações de outras classes, são acessados diretamente pelo usuário da classe, pois os demais controles estão dentro de cada uma das respectivas classes.

Um exemplo de como utilizar essas classes pode ser dado pelo programa exemplo `ControleContas`:

### Código 65 – Classe para testar o Controle Financeiro

```
class ControleContas
{
    static void Main()
    {
        Console.WriteLine("Controle de Contas");
        Console.WriteLine(" ");

        Conta c1 = new Conta("Conta de luz", 100);
        c1.categoriaConta.setNome("Despesas da casa");
        c1.tipoPagamentoConta.setNome("PAGAMENTO");

        Conta c2 = new Conta("Conta de água", 60);
        c2.categoriaConta.setNome("Despesas da casa");
        c2.tipoPagamentoConta.setNome("PAGAMENTO");

        Conta c3 = new Conta("Salário", 1000);
        c3.categoriaConta.setNome("Salários");
        c3.tipoPagamentoConta.setNome("RECEBIMENTO");

        Console.WriteLine("CONTA\t\t\tCATEGORIA\t\tTIPO\t\tVALOR");
        Console.WriteLine("{0}\t\t{1}\t\t{2}\t\t{3}",
            c1.Nome, c1.categoriaConta.getNome(),
            c1.tipoPagamentoConta.getNome(),
            c1.RetornaSaldo());
        Console.WriteLine("{0}\t\t{1}\t\t{2}\t\t{3}",
            c2.Nome, c2.categoriaConta.getNome(),
            c2.tipoPagamentoConta.getNome(),
            c2.RetornaSaldo());
        Console.WriteLine("{0}\t\t{1}\t\t{2}\t\t{3}",
            c3.Nome, c3.categoriaConta.getNome(),
            c3.tipoPagamentoConta.getNome(),
            c3.RetornaSaldo());
    }
}
```

No programa, são criadas três instâncias, sendo o valor e a descrição passados pelo construtor e o Tipo e a Categoria, pelos *setters* das instâncias dentro de cada objeto. Da mesma forma, na saída para a tela, as informações são acessadas pelos métodos *getters*, exceto o Nome (com N maiúsculo), que vem de um método particular do C#:

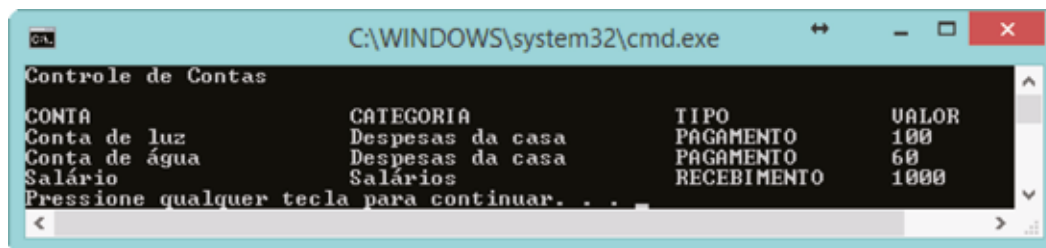


Figura 77 – Saída do controle financeiro

### 6.4.2 Herança

Herança é um conceito muito parecido com a delegação, que já vimos anteriormente. No entanto, seu objetivo é a derivação de classes, ou seja, a criação de uma classe (classe filha) onde sua base é uma classe já existente (classe pai), sendo adicionados a essa nova classe apenas atributos e métodos que não existam na classe original ou a execução de um método que seja diferente do método da classe original. Em outras palavras, a herança é um mecanismo de reaproveitamento em que as classes originais ficam contidas na nova classe.

A reutilização de classes via mecanismo de delegação é útil quando consideramos que a classe que reutiliza instâncias de outras classes é composta destas. Um bom exemplo é a classe *DataHora*, que é composta das classes *Data* e *Hora*, conforme vimos anteriormente.

Nem sempre o mecanismo de delegação é o mais natural para a reutilização de classes já existentes, embora seja simples. Em especial, quando queremos usar uma classe para servir de base à criação de outra mais especializada, a relação de composição imposta pelo uso do mecanismo de delegação acaba por criar soluções pouco naturais. Tal recurso é utilizado sempre que há necessidade de definir uma nova classe que seja um caso particular da classe original.

A solução é a reutilização de classes por meio do mecanismo de herança, no qual podemos criar uma classe usando outra como base e descrevendo ou implementando as diferenças e adições da classe usada como base, isso com a reutilização dos campos e métodos não privados da classe base. O mecanismo de herança é o mais apropriado para criar relações "é-um" entre classes.

No jargão das linguagens OO, a classe base, aquela que serve para a criação de classes mais especializadas, é chamada de superclasse (ou classe pai, ou classe base) e a que herda as características da superclasse é chamada de subclasse. Em UML, a relação de herança é representada por uma seta:

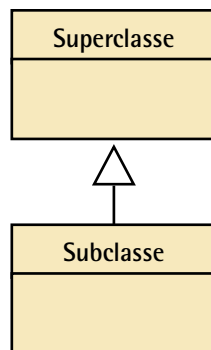


Figura 78 – UML do mecanismo de herança entre as classes



## Lembrete

Herança não é cópia, é uma derivação! Alterações realizadas na classe Pai poderão afetar a classe Filha, porém a classe Filha poderá ser definida com comportamentos distintos da classe Pai.

Uma das principais características da herança é a reutilização do código de uma determinada classe em outras classes. Herança é um conceito-chave usado na programação orientada a objetos para descrever uma relação entre as classes. Por meio da herança, uma classe copia ou herda todas as propriedades, atributos e métodos de uma outra classe, podendo assim estender sua funcionalidade.

A classe que cede os membros para a outra é chamada superclasse, classe pai ou classe base. A classe que herda os membros da outra classe é chamada subclasse ou classe derivada.

A herança permite a reutilização de código e especifica um relacionamento de especialização/generalização do tipo "é um".

Em C#, a identificação de herança acontece colocando o nome da superclasse logo após a declaração da classe, que será a subclasse:

```
class Filho : Pai
```

Nome da superclasse

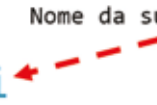


Figura 79 – Declaração da herança

Para entender o funcionamento do mecanismo de herança, vamos montar o processo passo a passo. Inicialmente, vamos construir uma classe simples chamada Filho, com um construtor que mostrará o texto "construtor Filho", e um método chamado Junior, que mostrará na tela "sou o método Junior":

### Código 66 – Classe Filho

```
class Filho
{
    public Filho()
    {
        Console.WriteLine("construtor Filho");
    }
    public void junior()
    {
        Console.WriteLine("sou metodo Junior ");
    }
}
```

Vamos construir também a Program que irá instanciar a classe Filho e executar o método Junior:

### Código 67 – Classe Program

```
class Program
{
    static void Main(string[] args)
    {
        Filho f = new Filho();
        f.junior();
        Console.WriteLine("Executei Main");
    }
}
```

Por enquanto, não existe novidade: ao ser executado, funciona como qualquer programa visto até aqui, e o que temos é a saída apresentada a seguir:

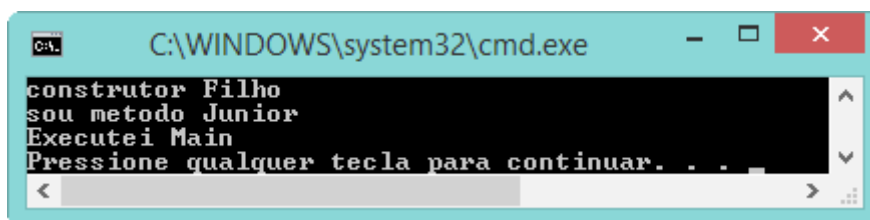


Figura 80 – Saída do programa simples

Uma instância *f* é criada usando *new*, que executa o construtor mostrando na tela que passou pelo construtor. A seguir, o método Junior da instância *f* é executado:



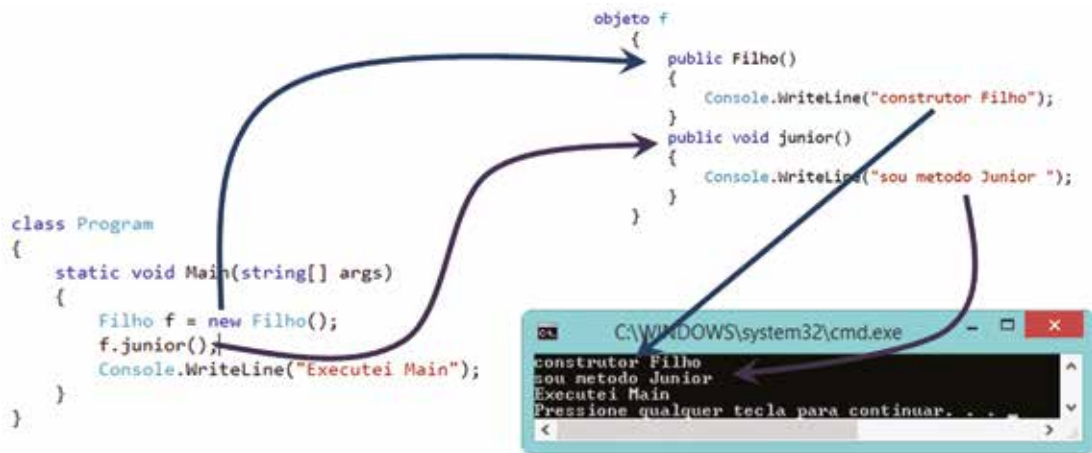


Figura 81 – Execução do programa simples

Vamos então construir a superclasse "Pai", assim como o "Filho". Há um aviso na tela que passou pelo construtor e um método que recebe uma *string* como parâmetro. Veja:

## Código 68 – Classe Pai

```
class Pai
{
    public Pai()
    {
        Console.WriteLine("construtor Pai");
    }
    public void papi(string nome)
    {
        Console.WriteLine("Superclasse pai, " + nome);
    }
}
```

A classe Filho será alterada para ser uma subclasse da classe pai:

### Código 69 – Classe Filho herdeiro do Pai

```
class Filho:Pai
{
    public Filho()
    {
        Console.WriteLine("construtor Filho");
    }
    public void junior()
    {
        Console.WriteLine("sou metodo Junior ");
    }
}
```

Assim, se fizermos uma Classe Program simples, na qual a classe Filho é instanciada, temos a saída mostrada na figura a seguir:

### Código 70 – Classe Filho sendo instanciada

```
class Program
{
    static void Main(string[] args)
    {
        Filho f = new Filho();
    }
}
```

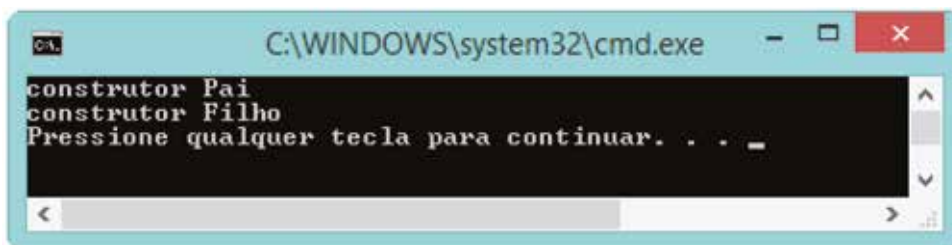


Figura 82 – Saída da classe Filho herdeira

O simples fato de instanciar uma classe Filho já consiste em mais de um processo. Para compreender, vamos à figura a seguir. A classe Filho, ao ser instanciada (1), pelo fato de ter a declaração de herança "Pai", instancia a classe Pai (2). Assim como todo o processo de instanciação, a classe Pai executa o construtor (3). Uma vez terminado o construtor, o comando é devolvido para a classe Filho, que só

então executa o seu construtor (4). Dessa forma, a saída (veja a figura anterior) mostra primeiro que o programa passou pelo construtor do Pai e, então, pelo construtor do Filho.

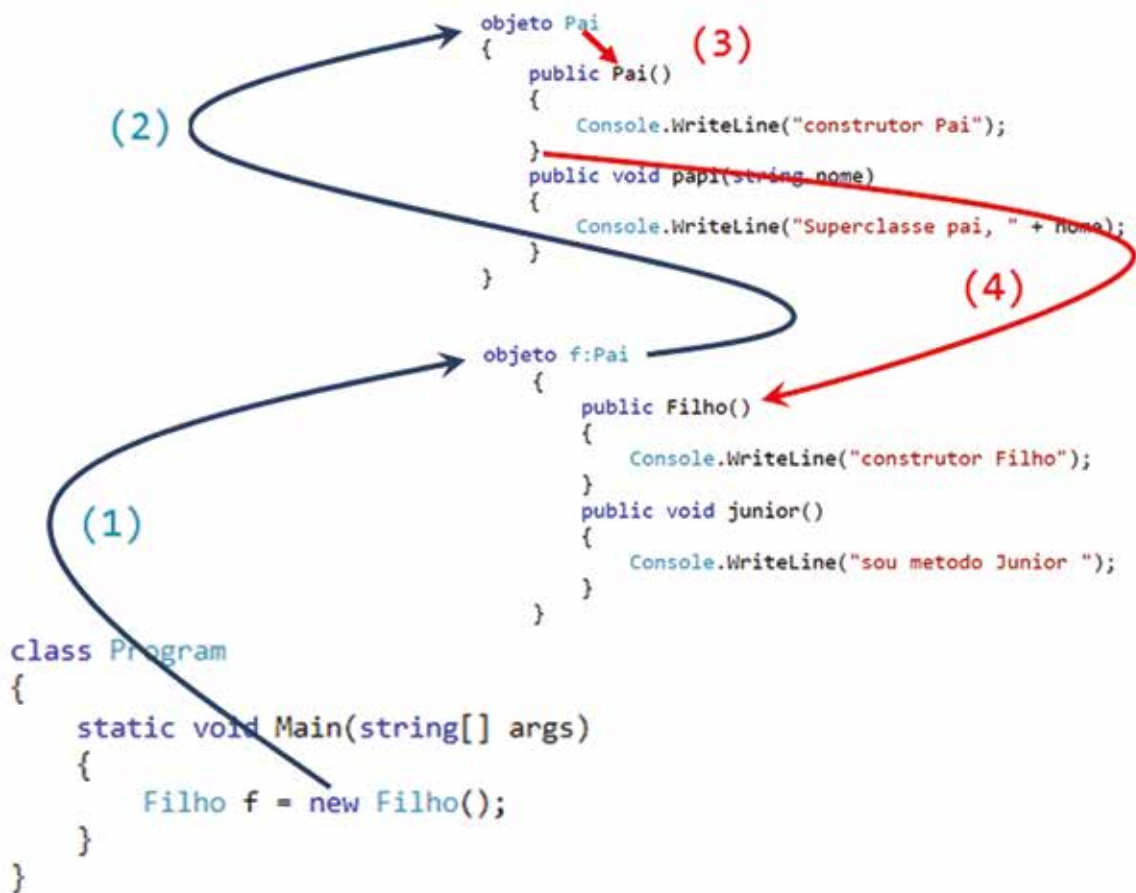


Figura 83 – Percurso da instanciação de uma classe herdeira

Uma vez compreendido o caminho seguido no processo de criar um objeto herdeiro, vamos ver a principal característica do mecanismo de herança, que é a reutilização dos atributos e métodos. Acrescentando-se a linha `f.papi("do Junior");` na classe `Program`, temos o código a seguir:

## Código 71 – Classe Program chamando um método herdado

```
class Program
{
    static void Main(string[] args)
    {
        Filho f = new Filho();
        f.papi("do Junior");
    }
}
```

O método `papí()` está na classe `Pai`; porém, para quem utiliza a instância da classe `Filho` é como se o método estivesse na própria classe `Filho`. Veja a saída do programa:

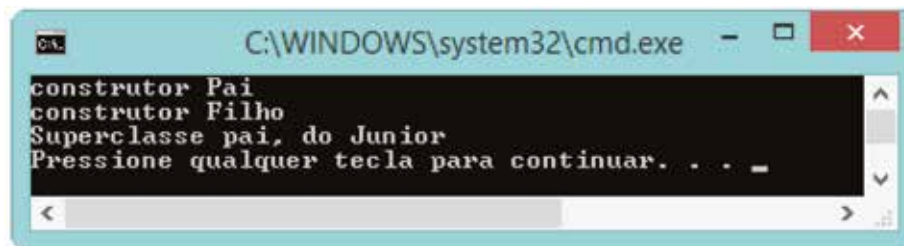


Figura 84 – Saída do programa usando um método herdado

### 6.4.3 Referência à superclasse

As classes derivadas ou subclasses podem ter acesso a métodos das superclasses usando a palavra-chave `base`. O acesso a métodos de classes ancestrais é útil para aumentar a reutilização de código. Se existem métodos na classe ancestral que podem efetuar parte do processamento necessário, devemos usar o código que já existe em vez de reescrevê-lo.

Algumas regras para o uso da palavra-chave `base` para chamar métodos de classes ancestrais como sub-rotinas são:

- Construtores são chamados simplesmente pela palavra-chave `base` seguida dos argumentos a serem passados para o construtor entre parênteses na declaração do construtor da subclasse, separado por *dois pontos* (`:`). Se não houver argumentos, a chamada deve ser feita como `:base()`.
- Métodos são chamados pela palavra-chave `base` seguida de um ponto e do nome do método. Se houver argumentos a serem passados para o método, estas devem estar entre parênteses, após o nome do método; caso contrário, os parênteses devem estar vazios.
- Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e, mesmo assim, somente se forem declarados na primeira linha de código do construtor da subclasse.
- Somente os métodos e construtores da superclasse imediata podem ser chamados usando a palavra-chave `base` – não existem construções como `base.base` que permitam a execução de métodos e construtores de classes ancestrais da classe ancestral. Caso seja necessário executar o construtor de uma classe ancestral da própria classe ancestral, os construtores podem ser escritos em cascata, de modo que o construtor da classe C chame o construtor da classe B que por sua vez chame o construtor da classe A.
- Se um método de uma classe ancestral for herdado pela classe descendente, ele pode ser chamado diretamente sem a necessidade da palavra-chave `base`.

Vamos tomar como exemplo o código classe Pai. Vamos colocar uma sobrecarga de construtor, conforme visto no código a seguir:

### Código 72 – Classe Pai com sobrecarga de construtor

```
class Pai
{
    public Pai()
    {
        Console.WriteLine("construtor Pai");
    }
    public Pai(string nome)
    {
        Console.WriteLine("Sou o construtor com parâmetros da classe Pai " + nome);
    }
    public void papi(string nome)
    {
        Console.WriteLine("Superclasse pai, " + nome);
    }
}
```

Na classe Filho, colocamos também uma sobrecarga de construtor; porém, esse construtor irá direcionar para a classe Pai passando como parâmetro o valor recebido no processo de criação da instância pela chamada da *base*.

### Código 73 – Classe Filho chamando a classe Pai

```
class Filho : Pai
{
    public Filho()
    {
        Console.WriteLine("construtor Filho");
    }
    public Filho(string nome)
        : base(nome)
    {
    }
    public void junior()
    {
        Console.WriteLine("sou metodo Junior ");
    }
}
```

Assim, ao criamos uma nova instância (veja o código a seguir), o programa procura o construtor que possui a assinatura *string* e esse construtor invoca a superclasse por meio da chamada à base, procurando também o construtor pai que possui a assinatura correspondente. Desse modo, é apresentada a saída que mostra o texto indicado pelo construtor cuja assinatura é uma *string* da superclasse, conforme ilustra a figura a seguir.

### Código 74 – Classe Program chamando o construtor com assinatura *string*

```
class Program
{
    static void Main(string[] args)
    {
        Filho f = new Filho("do junior");
    }
}
```

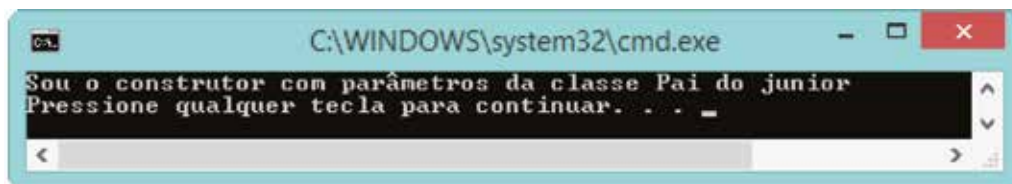


Figura 85 – Saída do programa com chamada para a classe base

#### 6.4.4 Sobreposição e ocultação

Os métodos das classes específicas têm prioridade sobre os métodos das classes genéricas. Se o método chamado existir na classe Filha, ele é que será chamado; se não existir, o método será procurado na classe Mãe.

Quando definimos um método com a mesma assinatura na classe base e em alguma classe derivada, estamos aplicando o conceito de reescrita de método, sobreposição ou superposição.

A razão de sobrepormos métodos é que métodos de classes herdeiras geralmente executam tarefas adicionais que os mesmos métodos das classes ancestrais não executam.

A declaração de campos em uma classe descendente com o mesmo nome de campos declarados na classe ancestral chama-se ocultação. Ao contrário da sobreposição de métodos, que é bastante útil em classes herdeiras, a ocultação de campos não oferece muitas vantagens e as poucas oferecidas podem facilmente ser implementadas por meio de métodos que retornam valores e são superpostos de acordo com a necessidade.

As principais regras de sobreposição de métodos e ocultação de campos e de uso de modificadores de acesso em classes herdadas são as seguintes:

- A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral – este pode ser acessado, de dentro da classe herdeira, com a palavra-chave *base*, contanto que não tenha sido declarado como *private*.

- Métodos declarados em uma subclasse com o mesmo nome mas assinaturas diferentes (por exemplo, número de argumentos diferentes) dos métodos da superclasse não sobrepõem esses métodos
- Métodos podem ser sobrepostos com diferentes modificadores de acesso, contanto que os métodos sobrepostos tenham modificadores de acesso menos restritivos. Em outras palavras, podemos declarar um método na superclasse com o modificador *private* e sobrepor esse método em uma subclasse com o modificador *public*, mas não podemos fazer o contrário.
- Métodos estáticos declarados em classes ancestrais não podem ser sobrepostos em classes descendentes, nem mesmo se não forem declarados como estáticos.
- Se um campo é declarado em uma superclasse e oculto em subclasses e se métodos que acessam esse campo são herdados, tais métodos farão referência ao campo da classe onde foram declarados.

Por padrão, as implementações dos métodos de uma superclasse não podem ser substituídas pelas subclasses. Para alterar esse padrão, devemos acrescentar o modificador *virtual* e *override* na que tem a mesma assinatura.

Para aplicar o conceito de sobreposição, vamos montar um programa com duas classes. A primeira classe armazena o nome e o salário de um empregado e retorna o valor do seu salário:

### Código 75 – Classe Empregado

```
class Empregado
{
    public string nome;
    private double SalarioBase;
    public Empregado(string nome, double SB)
    {
        this.nome = nome;
        this.SalarioBase = SB;
    }
    public virtual double CalculaPagto()
    {
        return SalarioBase;
    }
}
```

A segunda classe é de um comissionado, ou seja, uma especialização da classe dos empregados, já que além de receber um salário ele recebe uma comissão extra. Por essa configuração, não é necessário criar uma classe com todas as informações de um empregado, bastando então utilizar a classe Empregado como superclasse da classe Comissionado:



## Código 76 – Classe Comissionado

```
class Comissionado : Empregado
{
    private double comissao;
    public Comissionado(string nome, double SB,
        double comissao)
        : base(nome, SB)
    {
        this.comissao = comissao;
    }
    public override double CalculaPagto()
    {
        return base.CalculaPagto()+comissao;
    }
}
```

Ambas as classes possuem o método `CalculaPagto()`, se considerarmos sem as palavras-chaves *virtual* e *override*. Para uma classe externa, haveria um sério problema. Como a classe `Comissionado` tem o método `CalculaPagto()` e ele estende a classe `Empregado`, que também tem o mesmo método com a mesma assinatura, a visão externa seria de uma classe com dois métodos iguais. Dessa forma, para uma referência externa, utilizam-se as palavras *virtual* para o método sobreposto e *override* para o método superposto. A linguagem C# não obriga o uso das palavras sem resultar em erro, porém o editor avisa (*warning*) que existe um problema.

Ao executarmos o programa criando a instância para um empregado normal, o método `CalculaPagto()` é o da classe `Empregado`, resultando apenas no salário base:

## Código 77 – Program para uma instância de empregado normal

```
class Program
{
    static void Main(string[] args)
    {
        Empregado e1 = new Empregado("Maria", 1000);
        Console.WriteLine("{0}={1}", e1.nome, e1.CalculaPagto());
    }
}
```

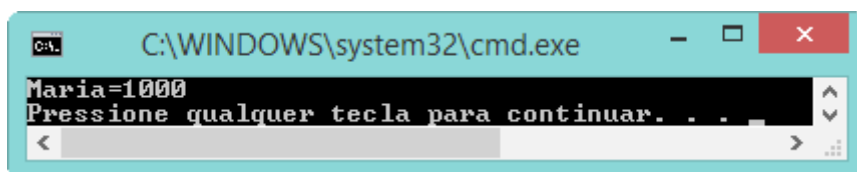


Figura 86 – Saída do cálculo do empregado simples



Alterando a classe Program para criar a instância de um comissionado, o `CalculaPagto()` executado é o da classe `Comissionado`, e não da classe `Empregado` (que é utilizado apenas para calcular o salário base). Dessa forma, na saída do programa o salário calculado é a soma do salário base com a comissão:

## Código 78 – Program para um empregado comissionado

```
class Program
{
    static void Main(string[] args)
    {
        Comissionado c1 = new Comissionado("Paula", 1000, 500);
        Console.WriteLine("{0}={1}", c1.nome, c1.CalculaPagto());
    }
}
```

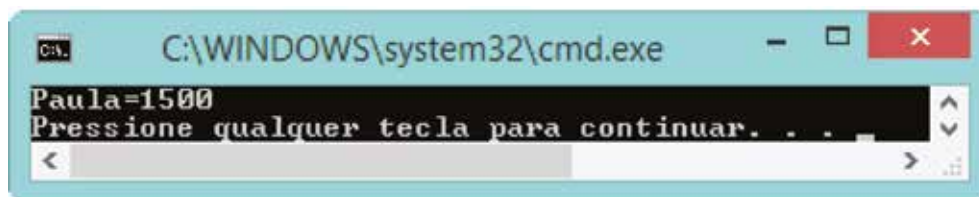


Figura 87 – Saída do cálculo do comissionado

Neste momento, temos completo o segundo pilar da POO, a herança.

## 6.5 Polimorfismo

O polimorfismo está relacionado com o conceito de herança, especificamente em relação a métodos. O mecanismo de herança permite a criação de classes a partir de outras já existentes com relações "é-um-tipo-de", de forma que, a partir de uma classe genérica, classes mais especializadas possam ser criadas.

Quando uma nova classe necessita de todos os atributos e métodos de uma já existente, porém a nova classe possui a execução de um ou mais métodos diferenciados, é possível herdarmos todos os métodos e atributos da classe original e realizar a alteração do comportamento do método somente na nova.

A partir desse momento, para todo objeto que se utilizar da nova classe e executar o método em questão, será executado o método novo, e não o original. O polimorfismo permite a manipulação de instâncias de classes que herdam de uma mesma classe ancestral de forma unificada: podemos escrever métodos que recebam instâncias de uma classe C, e os mesmos métodos serão capazes de processar instâncias de qualquer classe que herde da classe C, já que qualquer classe que herde de C é-um-tipo-de C. Desta forma, duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que

têm o mesmo nome mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse.

Polimorfismo, ter muitas formas. Em termos de programação, muitas formas significa que um único nome pode representar um código diferente, selecionado por algum mecanismo automático. Assim, o polimorfismo permite que um único nome expresse muitos comportamentos diferentes (SINTES, 2002, p. 122).

Vamos montar um exemplo de um método polimórfico. Para conceder um empréstimo, um banco tem as seguintes regras:

- Caso a pessoa que solicita o empréstimo for um cliente normal, o valor máximo do empréstimo é o dobro do seu limite de crédito.
- Se o solicitante for um funcionário do banco, o valor máximo é cinco vezes o valor do salário.
- No caso específico dos gerentes, que são funcionários mais qualificados, eles podem receber um empréstimo de dez vezes o valor do seu salário.

Usando o diagrama de classes do UML, vamos montar o relacionamento entre as classes das pessoas que podem solicitar empréstimo.

Como todos são pessoas físicas, vamos criar uma superclasse Pessoa com as informações comuns a todas as pessoas. No nosso caso, para facilitar, utilizaremos apenas o seu nome, mas poderia se colocar a data de nascimento ou a filiação, dentre várias possíveis.

A classe Pessoa pode ser particularizada em outras categorias. No nosso caso, o cliente é uma pessoa que possui um limite de crédito. Um funcionário é uma pessoa que tem salário e o gerente é um funcionário que dirige um departamento. Assim, como temos vários níveis de especialização, do mais geral (a pessoa) até o mais específico (o gerente), o relacionamento entre as classes será de herança:

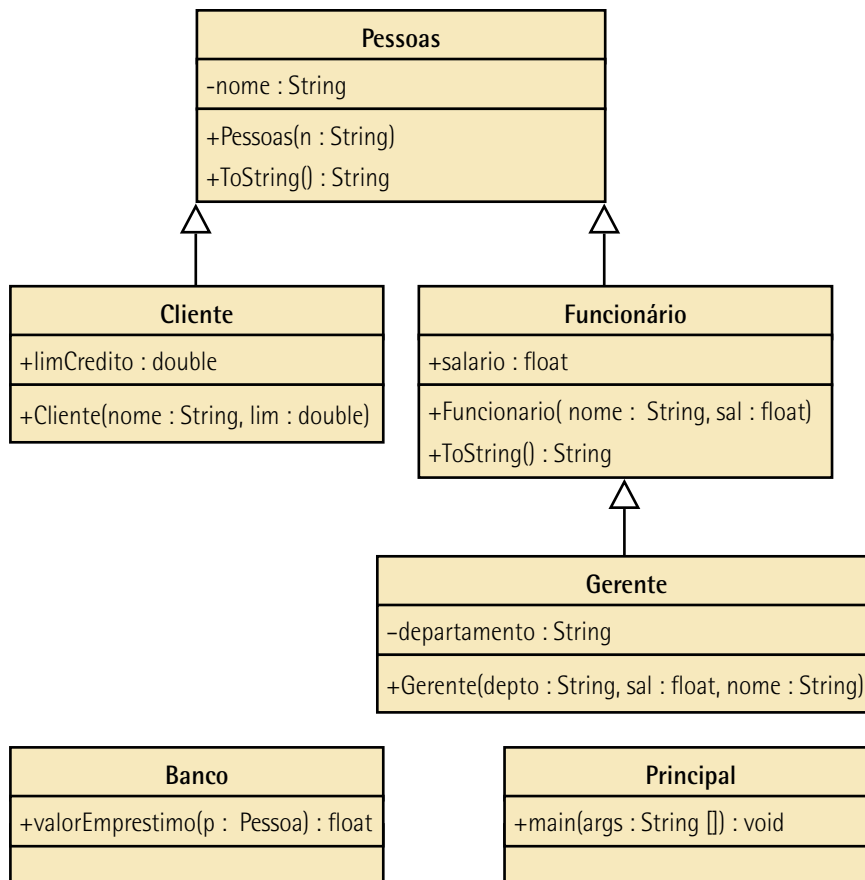


Figura 88 – Diagrama de classes para montar o método para calcular o valor do empréstimo

Estudamos anteriormente a conversão das classes em C#. A classe Pessoa (veja a figura a seguir) tem dois métodos, o construtor e um chamado *ToString()*. O método *ToString()* tem uma palavra em especial chamado *override* – veremos isso mais tarde, quando estudarmos as hierarquias de classes.

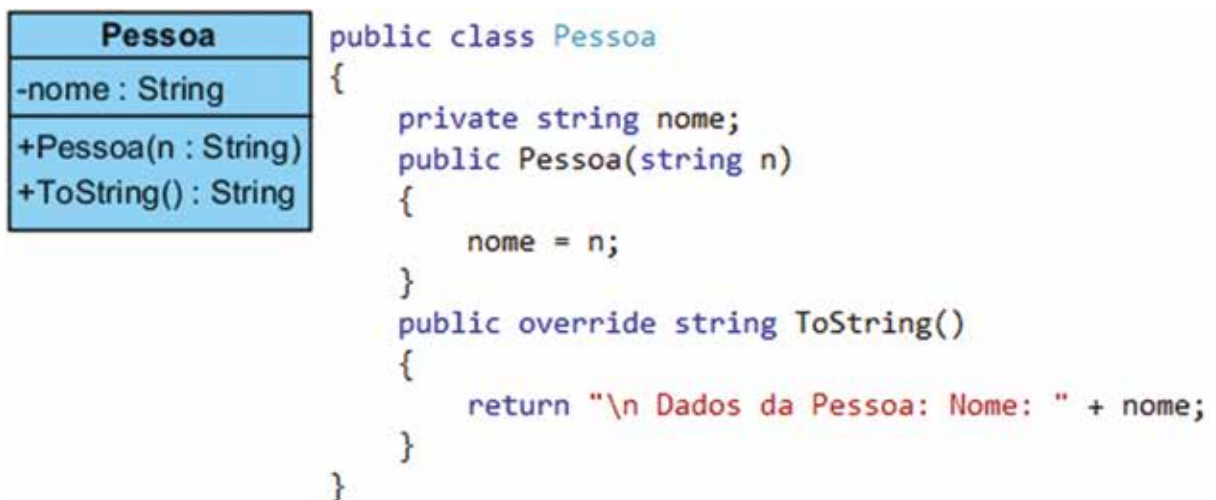
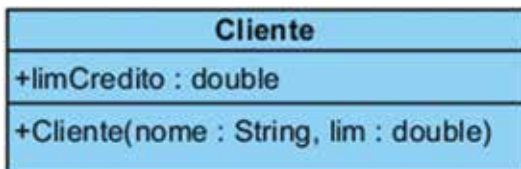


Figura 89 – Superclasse Pessoa

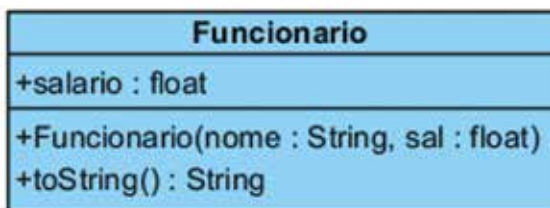
A classe Cliente (veja a figura a seguir) tem apenas o atributo `limCredito`, que armazena o valor do limite de crédito passado pelo construtor. As outras informações, por herança, ficam a cargo da superclasse Pessoa.



```
public class Cliente : Pessoa
{
    public double limCredito;
    public Cliente(string nome, double lim)
        : base(nome)
    {
        limCredito = lim;
    }
}
```

Figura 90 – Classe Cliente, herdeiro de pessoa

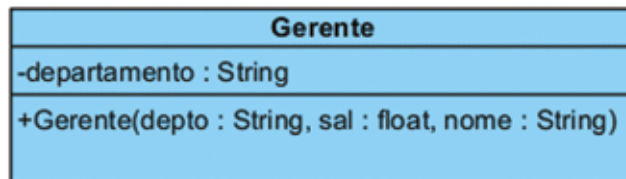
Assim como a classe Cliente, a classe Funcionario, vista na figura a seguir, é herdeira da classe Pessoa. Assim, ela só implementa o atributo salário, cujo valor é passado pelo construtor, e o nome é passado para o construtor da superclasse.



```
public class Funcionario : Pessoa
{
    public float salario;
    public Funcionario(string nome, float sal)
        : base(nome)
    {
        salario = sal;
    }
    public override string ToString()
    {
        return "\n\nFuncionario:" + base.ToString() +
            "\n Salario: " + salario;
    }
}
```

Figura 91 – Classe Funcionário, também herdeira de Pessoa

A classe Gerente, vista na próxima figura, é herdeira de Funcionário. Isso quer dizer que ele especifica a classe Funcionário, que já é uma especialização de Pessoa. Dessa forma, o construtor recebe como parâmetro o nome, que é atributo da classe Pessoa; o salário, que é atributo de Funcionario; e o departamento, que é um atributo da própria classe.



```
public class Gerente : Funcionario
{
    private string departamento;
    public Gerente(string depto, float sal, string nome)
        : base(nome, sal)
    {
        departamento = depto;
    }
}
```

Figura 92 – Classe Gerente: neta da Classe Pessoa e filha da classe Funcionario

A classe Banco (veja a figura a seguir) requer uma atenção especial. É nela que está o método polimórfico valorEmpréstimo.



```
public class Banco
{
    public static float valorEmprestimo(Pessoa p)
    {
        float valor;
        if (p is Gerente)
        {
            valor = ((Gerente)p).salario * 10;
        }
        else if (p is Funcionario)
        {
            valor = ((Funcionario)p).salario * 5;
        }
        else if (p is Cliente)
        {
            valor = (float)((Cliente)p).limCredito * 2;
        }
        else
        {
            valor = 0;
            return valor;
        }
    }
}
```

Figura 93 – O método polimórfico valorEmprestimo

O método `valorEmprestimo` recebe como parâmetro a variável `p`, que é um tipo da classe `Pessoa`.

```
public static float valorEmprestimo(Pessoa p)
```

Portanto, qualquer objeto da classe `Pessoa` e seus descendentes serão aceitos como parâmetro.

Quando vimos o modelador `cast`, observamos que ele tem um operador que verifica se um objeto é uma instância de uma classe. No programa, ao executar a condicional:

```
if (p is Gerente)
```

verifica-se que o objeto `p` é da classe `Gerente`.

O `cast` também é utilizado para modelar a variável `p`. Como `p` foi declarado como sendo do tipo `Pessoa`, ou seja, uma superclasse, ele não conhece a sua descendência. Assim, ao fazer o `cast` no `p`,

```
valor = ((Gerente)p).salario * 10;
```

o programa reconhece que `p` é da classe `Gerente`. O mesmo é feito para as opções de `Funcionario` e `Cliente`.

Para testarmos o polimorfismo do método `valorEmprestimo`, vamos executar algumas vezes o programa alterando as instâncias entre as diversas classes. O objetivo é verificar se o método `valorEmprestimo` pode receber diferentes tipos como parâmetro e funcionar sem problemas, devolvendo o valor correto.

No primeiro teste, visto no código a seguir, cria-se uma instância, `f1` (lembrando que na verdade é uma referência, mas aqui, para facilitar, assumimos como ele sendo a própria instância), de um funcionário. Essa instância é passada para o método `valorEmprestimo()`.

### Código 79 – Testando para um objeto da classe `Funcionario`

```
class Program
{
    public static void Main(string[] args)
    {
        Funcionario f1 = new Funcionario("Joao", 3000);
        Console.WriteLine(f1.ToString() +
            "\n Empréstimo: " + Banco.valorEmprestimo(f1));
    }
}
```

Ao executar, verificamos que o método recebeu sem problema um objeto que é uma instância da classe `Funcionario`, conforme a figura a seguir. Sendo um `Funcionario` que tem como salário R\$ 3000,00, ele tem direito a um empréstimo de cinco vezes o seu salário. Portanto, o cálculo resulta em R\$ 15.000,00.

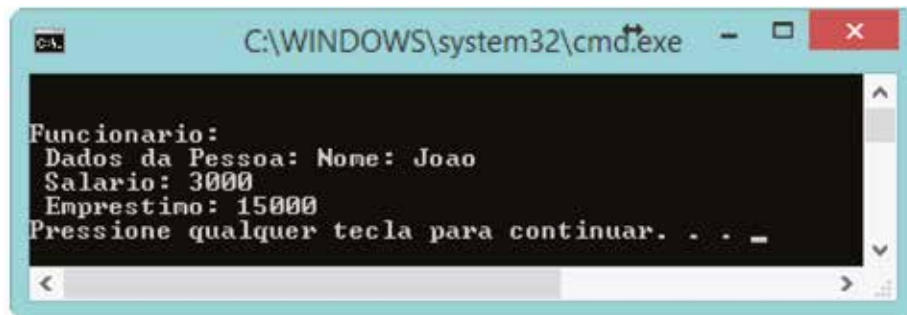


Figura 94 – Saída do cálculo do empréstimo para `Funcionario`

O segundo teste é para outra instância de `Funcionario`, agora sendo referenciado como do tipo `Pessoa`:

## **Código 80 – Testando para um objeto a classe `Funcionario`, porém, sendo declarado como uma `pessoa`**

```
class Program
{
    public static void Main(string[] args)
    {
        Pessoa f2 = new Funcionario("Jose", 4000);
        Console.WriteLine(f2.ToString() +
            "\n Empréstimo: " + Banco.valorEmprestimo(f2));
    }
}
```

Como o funcionário `f2` recebe R\$ 4000,00, ele deverá ter direito a receber R\$ 20.000,00 de empréstimo. Com o teste anterior, foi verificado que o método não tem problema com objetos de classe `Funcionario`:

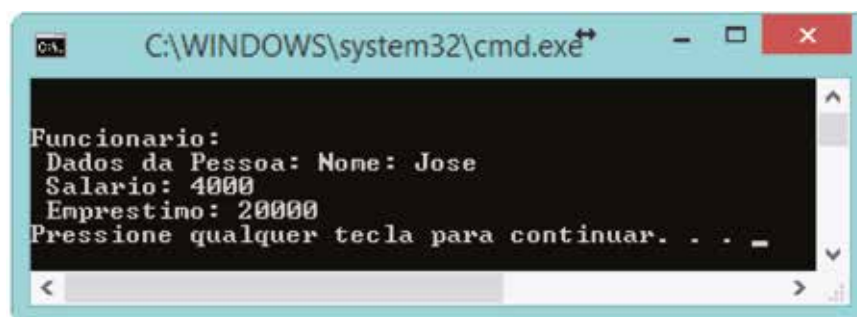


Figura 95 – Saída para o outro objeto da classe `Funcionario`



Agora vamos fazer o teste com uma instância de outra classe. O objeto g1 é da classe gerente e será passado como parâmetro para o método valorEmprestimo:

### Código 81 – Testando para a uma instância da classe Gerente

```
class Program
{
    public static void Main(string[] args)
    {
        Gerente g1 = new Gerente("Contabilidade", 5000, "Marcos");
        Console.WriteLine(g1.ToString() +
            "\n Emprestimo: " + Banco.valorEmprestimo(g1));
    }
}
```

O valor do empréstimo para gerente é de dez vezes o valor do seu salário. Assim, como o salário é de R\$ 5.000,00, o programa mostra que o valor do empréstimo é de R\$ 50.000,00 (veja a figura a seguir). Assim, já se pode afirmar que o método valorEmprestimo é polimórfico, pois funcionou sem problemas tendo mais de um tipo diferente como entrada.

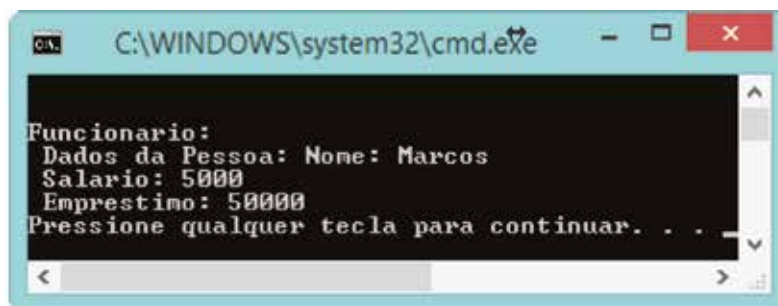


Figura 96 – Saída para instância de Gerente

Para completar o teste, vamos verificar com uma classe que não tem relação alguma com funcionário e gerente, pois o valor do empréstimo é relacionado com o valor do salário. O cliente tem direito de empréstimo ao dobro do valor do seu limite.

Uma instância da classe Cliente, c1, é criada tendo como valor do seu limite R\$ 3.500.00:

### Código 82 – Testando para um objeto da classe Cliente

```
class Program
{
    public static void Main(string[] args)
    {
        Cliente c1 = new Cliente("Maria", 3500);
        Console.WriteLine(c1.ToString() +
            "\n Emprestimo: " + Banco.valorEmprestimo(c1));
    }
}
```



Assim, o programa mostra que o valor do empréstimo é de R\$ 7.000,00:

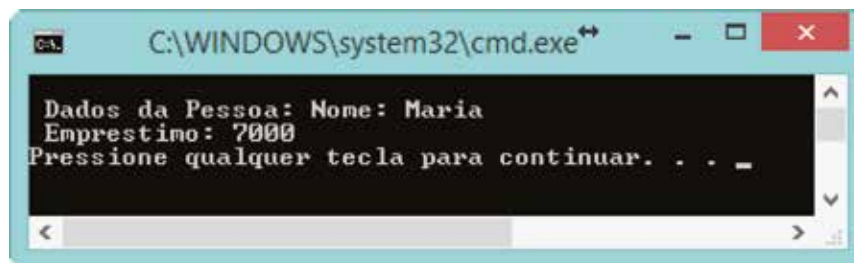


Figura 97 – Saída para objeto Cliente passado como parâmetro

Mostrou-se com isso que o método `valorEmpréstimo` funcionou sem problemas com entradas de objetos de diferentes tipos. Esta é, na prática, a característica do polimorfismo: a possibilidade de trabalhar com diferentes tipos e classes.

Encerramos assim os três pilares da POO: o encapsulamento, a herança e, neste item, o polimorfismo.



## Saiba mais

Para aqueles que pretendem desenvolver jogos, existe um poderoso programa para isso, o Unity 3D. Para o seu uso, é fundamental a noção de orientação a objeto, pois nele os conceitos de classes, objetos e herança são aplicados diretamente no desenvolvimento de jogos. Uma das linguagens para o desenvolvimento de *scripts* para o jogo é o C#. Existe uma versão gratuita para aqueles que gostariam de iniciar o seu aprendizado em:

UNITY. *Create the games you love with Unity*. [s.d.]. Disponível em: <<http://unity3d.com/unity>>. Acesso em: 29 out. 2014.

## 6.6 Objetos

Assim como uma variável, uma classe, ao ser instanciada, se torna um objeto; este, por sua vez, se torna residente em memória.

Tecnicamente, quando os objetos estão instanciados, trabalhamos com sua referência em memória, não com o objeto em si. Esse fato diferencia os objetos das variáveis (ou tipos primitivos, tais como *double* e *int*), pois nas variáveis trabalhamos com o valor que realmente está em memória.

Na maioria das linguagens de programação, todo objeto, uma vez instanciado, deve ser destruído, ou seja, retirado da memória. Porém, plataformas de programação mais modernas possuem um conceito chamado *garbage collector* (coletor de lixo), que retira da memória objetos não mais referenciados automaticamente.



### Observação

Os conceitos de encapsulamento, herança, polimorfismo e controle de objetos serão detalhados mais adiante.



### Resumo

Nesta unidade vimos os principais conceitos da programação orientada à objetos e a sua programação utilizando a linguagem C#. Conhecemos a construção de uma classe, o seu processo de transformar em objeto e os relacionamentos entre as classes. Vale ressaltar que cada classe corresponde a uma entidade em nosso sistema com um objetivo único e claro e que a classe e o objeto são coisas distintas: a classe é a definição da entidade e o objeto é a implementação alocada em memória que será utilizada efetivamente pelo sistema. Vimos também que o procedimento de criar objetos (*new*) é chamado de instanciar objetos.

Nesta unidade também foram abordados conceitos como atributos, as características de uma classe e a sua ação por meio dos métodos. Vimos os métodos especiais como o construtor, que inicializa uma classe, e a sobrecarga, na qual métodos de mesmo nome podem estar numa mesma classe desde que tenham diferentes assinaturas e o nome e a sequência de tipos dos parâmetros também diferentes. Com os métodos, vimos também a técnica para fazer o encapsulamento e o isolamento dos atributos dentro de uma classe.

Encerramos vendo como as classes se relacionam com as associações, em que uma classe utiliza outra como atributo, e a herança, em que uma classe é criada usando outra como base. A utilização de herança gera hierarquia de classes, ou seja, a estrutura de quais são as superiores (ou classes Pai) e quais são as subclasses (ou classes filho). Graças à herança e à sobrecarga, é possível haver o polimorfismo, em que um mesmo método consegue tratar vários tipos de informação.



### Exercícios

**Questão 1.** (TRT/SC 2013) Na programação orientada a objetos, as classes podem conter, dentre outros elementos, métodos e atributos. Os métodos:

- A) devem receber apenas parâmetros do mesmo tipo.
- B) não podem ser sobrecarregados em uma mesma classe.
- C) precisam possuir corpo em interfaces e classes abstratas.
- D) podem ser sobrescritos em aplicações que possuem relação de herança.
- E) definidos como `private` só podem ser acessados de classes do mesmo pacote.

Resposta correta: alternativa D.

### Análise das alternativas

A) Alternativa incorreta.

Justificativa: os métodos podem receber quantos e quaisquer tipos de parâmetros que forem necessários. Essa obrigatoriedade proposta pela opção é equivocada.

B) Alternativa incorreta.

Justificativa: os métodos podem ser sobrecarregados na mesma classe, se necessário. Um exemplo é a possibilidade de sobrecarregarmos o método construtor de uma classe com várias possibilidades de construção de um objeto de uma classe.

C) Alternativa incorreta.

Justificativa: pelo contrário, as classes interface e abstratas só podem possuir a assinatura dos métodos (cabeçalhos). A implementação desses métodos deve ser feita nas classes que utilizaram a classe interface ou que "herdaram" os métodos das classes abstratas.

D) Alternativa correta.

Justificativa: os métodos de uma subclasse herdados de superclasses podem ser sobrescritos. Um exemplo disso é a possibilidade de re-escrever algum método da superclasse de forma mais condizente para as necessidades da subclasse filha.

E) Alternativa incorreta.

Justificativa: os métodos declarados como `private` só podem ser acessados, modificados ou executados por métodos da própria classe, sendo completamente ocultos por outras classes (mesmo herdeiras ou derivadas) ou pacote de classes objeto.

**Questão 2.** (DPE/RJ 2014) Considere o código escrito na linguagem C# mostra a seguir:

```
using System.IO;
using System;

public class Veiculo
{ public virtual void mover()
  { Console.Write("Movendo");
  }
}

public class Automovel:Veiculo
{ public override void mover()
  { Console.Write("Acelerando");
  }
}

public class Fusca:Automovel
{ public override void mover()
  { Console.Write ("Passeando");
  }
}

class Program
{ static void Main()
  { Veiculo veiculo = new Fusca();
    veiculo.mover();
  }
}
```

O resultado produzido pela execução desse código é:

- A) Acelerando
- B) Passeando
- C) Movendo
- D) MovendoAcelerandoPasseando
- E) AcelerandoPasseando

**Resolução desta questão na plataforma.**