

Unidade IV

INTERAÇÃO E COMUNICAÇÃO ENTRE OBJETOS

Esta unidade apresentará alguns conceitos avançados da POO em relação à interação entre objetos.

7 INTERAÇÃO ENTRE OBJETOS

7.1 Comunicação e associação

Associação entre objetos diz respeito ao ato de um objeto utilizar outro para realizar alguma operação. É um recurso simples e utilizado a todo o momento.

Conforme vimos anteriormente, associação não é a mesma coisa que composição: associação é simplesmente o uso de um objeto diretamente, conforme o contexto do projeto em que ele estiver sendo aplicado.

7.2 Classes abstratas e interfaces

Como já ilustrado, umas das premissas do POO é a abstração. Geralmente, as linguagens de programação permitem o uso de classes abstratas para a definição de estrutura, de modo a padronizar a criação de outras classes, por meio da herança. Vimos que o mecanismo de herança é uma poderosa ferramenta que permite a reutilização de código por meio da criação de classes baseadas em outras classes já existentes. Uma característica do mecanismo de herança é que deve existir uma superclasse a partir da qual as subclasses serão criadas, sendo que tanto a superclasse quanto as subclasses podem ser usadas para criar instâncias para uso em outras classes e aplicações.

Com o mecanismo de herança visto até agora, devemos criar uma classe ancestral que tenha os campos e métodos comuns a todas as suas herdeiras. Devemos também fazer a implementação dos métodos de forma que instâncias da classe ancestral possam ser criadas. Nem sempre isso é desejável – em alguns casos seria interessante descrever os campos e métodos que as classes herdeiras devem implementar, mas não permitir a criação de instâncias da classe ancestral. Dessa forma, a classe ancestral passaria a ser somente um guia de quais métodos e campos devem ser implementados nas classes herdeiras. Em outras palavras, a classe ancestral ditaria para as classes descendentes o que deve ser feito, mas sem necessariamente dizer como deve ser feito.

São mais comuns dois mecanismos que permitem a criação de classes que somente contenham descrições de campos e métodos que devem ser implementados, mas sem efetivamente implementar esses métodos. Classes que declaram, mas não implementam métodos, são particularmente úteis na criação de hierarquias de classes, porque não permitem a criação de instâncias delas e exigem que as classes descendentes implementem os métodos nelas declarados. Os dois mecanismos são as classes abstratas e as interfaces.

7.2.1 Classes abstratas

É um dos mecanismos de criação de superclasses com declarações, mas sem a definição de todos os métodos, permitindo, porém, a criação de métodos declarados como abstratos. Esses métodos abstratos são somente declarações como um método normal com seu nome, modificadores, tipo de retorno e lista de argumentos, mas não tendo um corpo que contenha os comandos da linguagem que esse método deva executar precedidos da palavra *abstract*. Se uma classe declara um método abstrato, as classes que herdarem dela deverão obrigatoriamente implementar o método abstrato com a mesma assinatura declaradas na classe ancestral.

Como visto, os métodos abstratos são declarados com o modificador *abstract*. Se uma classe tiver algum método abstrato, ela também deverá obrigatoriamente ser declarada com o modificador *abstract*, mas podem haver classes abstratas sem nenhum método abstrato. Reforçamos que uma classe herdeira de uma classe abstrata é obrigada a implementar todos os métodos declarados como abstratos na classe ancestral.

Métodos abstratos não podem ter corpo (a parte entre chaves). Somente a declaração do método é necessária, mas mesmo que variáveis passadas como argumentos nunca sejam usadas, seus nomes devem ser especificados.

Construtores de classes abstratas não podem ser abstratos. Mesmo que a classe abstrata não possa ser instanciada, seus construtores podem iniciar os campos da classe que serão usados por subclasses, sendo imprescindíveis em praticamente todos os casos.

A declaração de herança de uma classe abstrata na subclasse segue a mesma regra de formação de uma herança normal.



Observação

Uma classe abstrata não pode ser instanciada diretamente, pois se fosse possível aconteceriam problemas, já que faltariam as implementações dos métodos abstratos. Assim, a instanciação de uma classe abstrata somente ocorre a partir das suas classes herdeiras.

Em UML, uma classe abstrata é identificada com o nome da classe escrita em itálico.

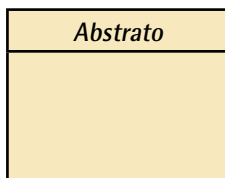


Figura 98 – Representação de uma classe abstrata

Vamos tomar como exemplo uma classe abstrata que implementa um botão. Um botão dentro da programação tem vários atributos e métodos implementados, mas essa classe exemplo tem como características a possibilidade de colocar um rótulo e estar ativa ou desativa. Como é uma classe abstrata, ela deixa a implementação da atividade para as suas classes herdeiras.

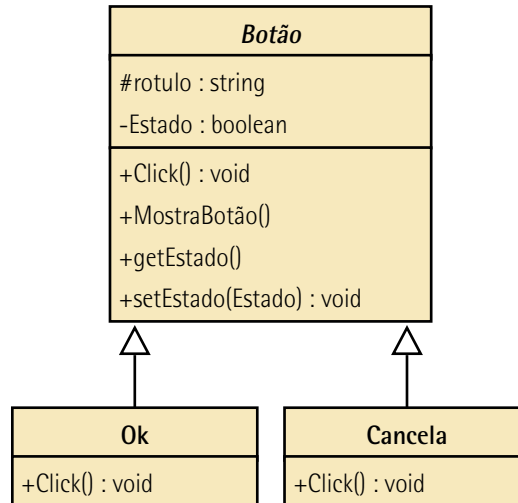


Figura 99 – Exemplo de uma classe abstrata botão

Nesse exemplo, temos duas classes herdeiras, o botão Ok, que ao ser clicado efetua um cálculo, e o botão de Cancela, que termina o processo iniciado.

```
abstract class Botao
{
    protected string rotulo;
    private bool estado;

    public bool Estado
    {
        get { return estado; }
        set { estado = value; }
    }
    abstract public void Click();
    public void MostraBotao()
    {
        if (this.estado)
            Console.WriteLine("Botao {0} Ativo", this.rotulo);
        else
            Console.WriteLine("Botao {0} Inativo", this.rotulo);
    }
}
```

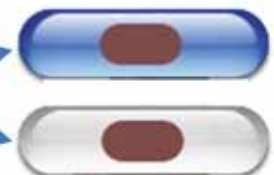


Figura 100 – Classe abstrata Botão

A classe abstrata Botão tem dois atributos. O atributo rótulo irá escrever no corpo do botão o texto definido pela classe herdeira. O segundo atributo define o estado do botão (se o botão estará ativo ou não).

Note que o atributo estado é do tipo *bool*, ou seja, ele recebe os valores *true* ou *false*. Temos um método abstrato que fará com que seja obrigatória a implementação de uma ação no herdeiro quando o botão for acionado, conforme vimos na figura anterior. Temos também um método concreto `Mostrabotão()`, que mostra na tela o estado do botão. Esse método, sendo concreto, não precisa ser implementado na classe herdeira, portanto ela faz parte do comportamento padrão que todo botão irá ter.

Como vimos, vamos criar a classe de dois botões. A primeira será o botão Ok – o construtor irá definir o rótulo e deixar o botão ativo. O método que irá atuar quando o botão for acionado – no nosso caso, uma simulação de um cálculo, mostrando "calculando" na tela – na realidade pode ser um programa bem complexo. Esse método não é facultativo, pois sua implementação está sendo imposta pelo método abstrato da superclasse:

```
class Ok : Botao
{
    public Ok()
    {
        base.rotulo = "Ok";
        Estado = true;
    }
    public override void Click()
    {
        if (Estado)
            Console.WriteLine("Calculando");
    }
}
```




Figura 101 – Implementação da classe do botão Ok

A segunda classe herdeira será a do botão Cancelar. Esse botão receberá o rótulo "Cancelar" e iniciará inativo. O seu método `Click()` irá simular as operações que vierem a ser feitas quando o botão for acionado (no nosso caso, mostrando na tela "Cancelando", conforme a figura a seguir). Note que em ambas as classes o botão somente irá funcionar quando o seu estado estiver ativo (estado igual a *true*).

```
class Cancela : Botao
{
    public Cancela()
    {
        base.rotulo = "Cancelar";
        Estado = false;
    }
    public override void Click()
    {
        if (Estado)
            Console.WriteLine("Cancelando");
    }
}
```




Figura 102 – Implementação da classe do botão Cancelar

Para simular a execução do programa, inicialmente vamos criar uma instância do botão Ok e acionar (veja a figura a seguir). Assim, o método `Click` irá mostrar que o programa está calculando.

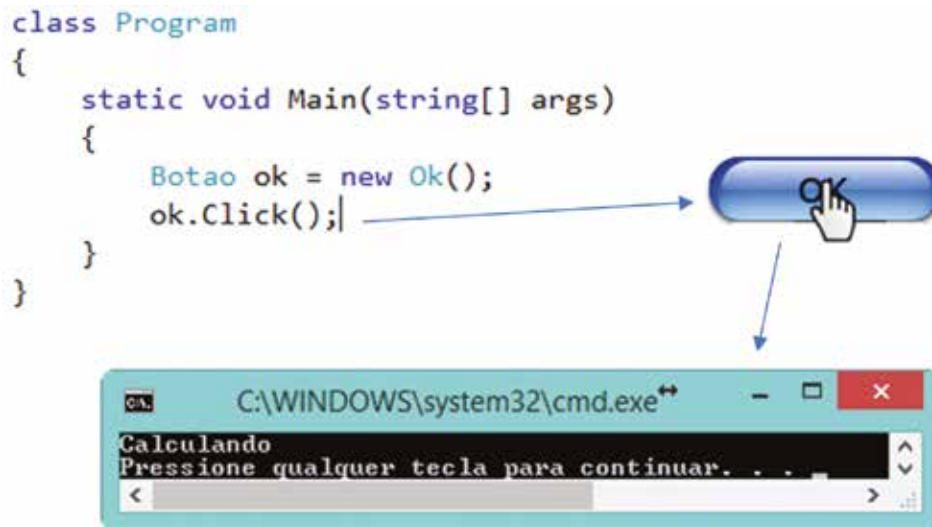


Figura 103 – Clicando o botão Ok

A segunda simulação se dá criando uma instância do botão cancela e acionando-a. Ao contrário do botão Ok, nada é mostrado na tela. Veja:

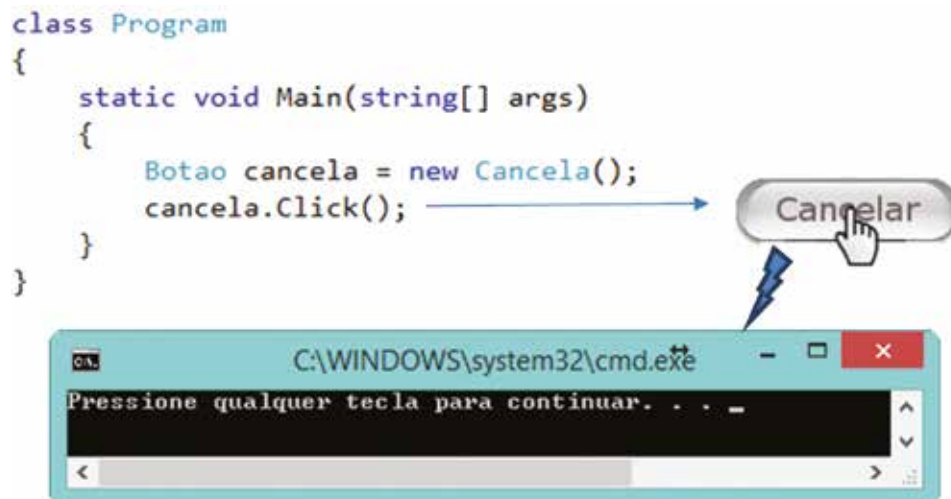


Figura 104 – Ao clicar o botão Cancelar nada acontece

Vamos verificar o estado do botão utilizando o método concreto `Mostrabotão()` que está na superclasse:

Para a classe que está utilizando a instância do botão Cancelar, o método `Mostrabotão()` faz parte da classe `Cancela`, pois ela é herdada da classe `Botão`. Assim, ao utilizar o método, vemos que o botão está desativado, conforme a figura:

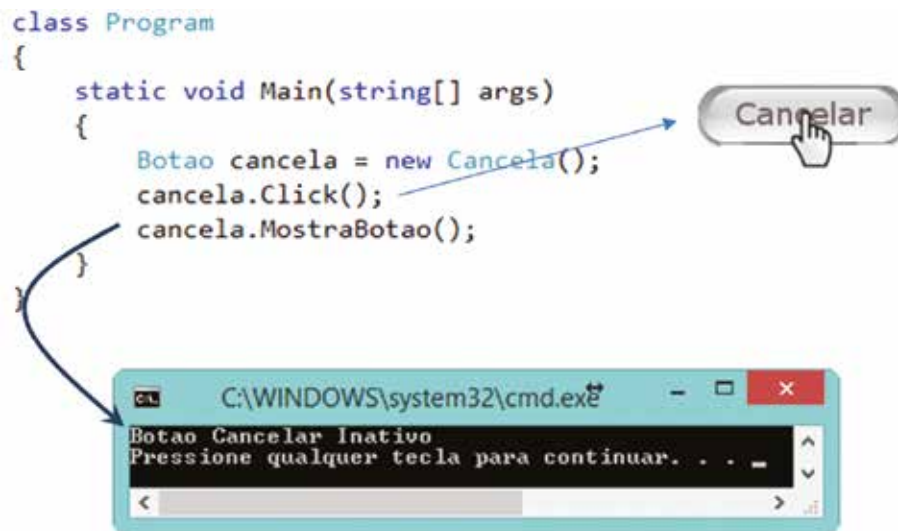


Figura 105 – Verificando o estado do botão Cancelar

Para ativar o botão Cancelar, mudamos o estado para *true*, novamente lembrando que o atributo estado não faz parte da classe Cancela, mas da superclasse Botão. Alterando o valor do atributo, utilizamos mais uma vez o método MostraBotao() para verificar se o botão foi ativado:

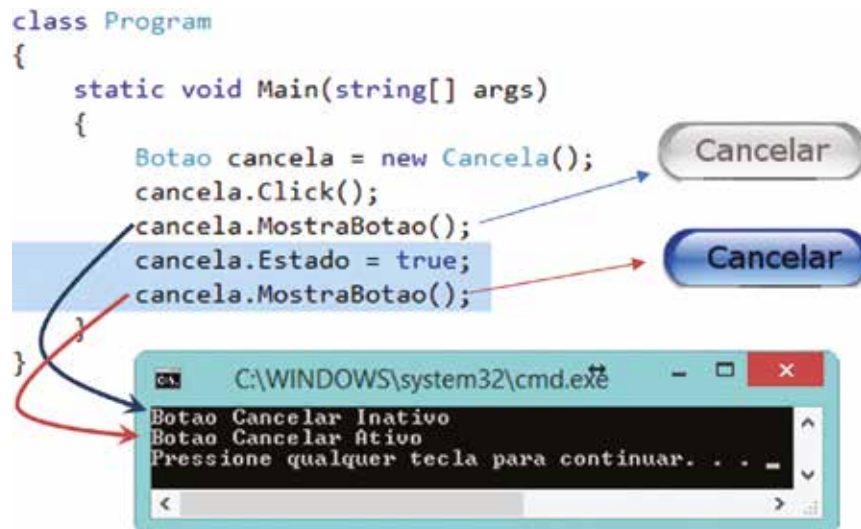


Figura 106 – Ativando o botão Cancelar e verificando o seu estado

Assim, o botão Cancelar fica ativo para que possamos acioná-lo, e na tela é mostrado que o método agora está trabalhando ("Cancelando").

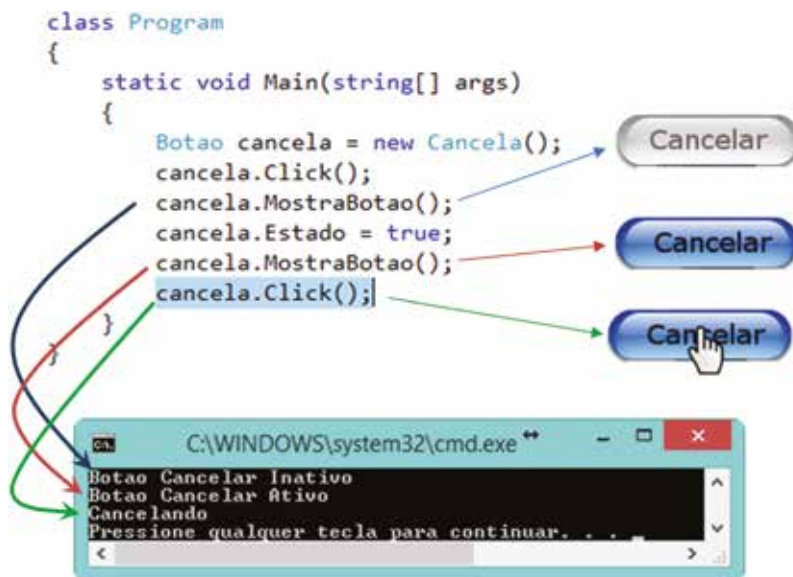


Figura 107 – Clicando após ativar o botão

De uma forma geral, podemos dizer que classes abstratas que fornecem guias para a montagem de classes concretas maiores funcionam como brinquedos de blocos de montar: cada classe abstrata fornece um pequeno conjunto de métodos e atributos específicos (com entradas e saídas de dados padronizados) para que sejam utilizados e combinados em uma classe que herdará esses conjuntos.

7.2.2 Interfaces

Classes abstratas podem conter métodos não abstratos que serão herdados e poderão ser utilizados por instâncias de classes herdeiras. Se a classe não tiver nenhum método não abstrato, podemos criá-la como uma interface, que segue um modelo de declaração diferente do usado para classes mas tem funcionalidade similar à de classes abstratas.

Todos os métodos na interface são implicitamente *abstract* e *public* e não podem ser declarados com seus corpos. Campos, se houverem, serão implicitamente considerados *static* e *final*, devendo, portanto, ser iniciados na sua declaração.

Qualquer classe pode implementar uma ou várias *interfaces* simultaneamente. *Interfaces*, portanto, podem ser consideradas um mecanismo simplificado de implementação de herança múltipla.

Outro uso interessante para as *interfaces* se dá ao implementar bibliotecas de constantes, pois os campos declarados em uma interface devem ser declarados como estáticos e inicializados. Assim, pode-se ter *interfaces* somente com campos e sem métodos, e qualquer classe que venha a implementá-la terá disponíveis os seus atributos. Desse modo, a técnica que possibilita uma maior padronização é a interface. Ela padroniza o formato da classe, ou seja, como deverão ser suas entradas e saídas de dados, deixando a implementação dos procedimentos a cargo de cada classe que herdou essas *interfaces*.



Observação

Ao contrário da linguagem Java, na qual o uso de atributos estáticos pode ser feito nas interfaces implementando bibliotecas de constantes, a linguagem C# apenas declara os métodos.

Uma interface declara apenas os métodos que obrigarão as suas classes a implementarem. Dessa forma, ela só tem declarações de métodos, sem possuir atributos e construtores. Os métodos declarados em uma interface podem ser sobrecarregados, ou seja, podem existir métodos com nomes diferentes mas com assinaturas diferentes.

Uma interface lista os serviços fornecidos por um componente. A interface é um contrato com o mundo exterior, que define exatamente o que uma entidade externa pode fazer com o objeto. Uma interface é o painel de controle do objeto (SINTES, 2002, p. 22).

Em UML a interface é representada com <<interface>> acima do nome da interface.

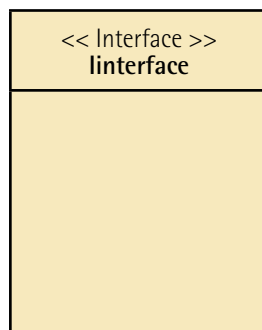


Figura 108 – Interface

O prefixo dos nomes de interface é por convenção feito com a letra "I" para indicar que o tipo é uma interface. Para garantir que a definição de um par de classe/interface em que a classe é uma implementação padrão da interface, os nomes só diferem pela letra "i" no prefixo do nome da interface.

Em C#, ela é implementada utilizando a palavra interface no lugar de *class*. No corpo, são feitas as declarações dos métodos que serão padronizados pela interface. Na classe que implementa a interface, são definidos os métodos impostos pela interface:

Código 83 – Implementação padrão de interface

```
interface IexemploInterface
{
    void MetodoExemplo();
    void MetodoExemplo1();
}

class ClasseImprementadora : IexemploInterface
{
    void IexemploInterface.MetodoExemplo()
    {
        // Implementação do método.
    }
    void IexemploInterface.MetodoExemplo1()
    {
        // Implementação do método.
    }
}
```

No processo de criação de uma interface, normalmente é criada uma referência do tipo da interface e a instância é feita utilizando a classe que implementa a interface:

Código 84 – Criando e utilizando uma classe implementada utilizando uma interface

```
class program
{
    static void Main()
    {
        IexemploInterface obj = new ClasseImprementadora ();
        obj.MetodoExemplo();
    }
}
```

Exemplo: controle remoto universal

Vamos neste exemplo criar um controle remoto universal que servirá para acionar uma televisão, um DVD player e um simples abajur. Como cada aparelho tem um comportamento diferente, os botões do controle deverão ter funções diferentes conforme o aparelho em que esse controle será utilizado. Assim, vamos abstrair o controle remoto com os seguintes botões: ligar, desligar, seta para cima (aumento de volume), seta para baixo (diminuição de volume), seta para a esquerda (retroceder) e seta para a direita (avançar).

O controle remoto sozinho não tem função alguma – é apenas uma série de botões que só tem função quando associado a alguma coisa. Abstraindo, podemos construir uma interface do controle remoto, conforme a figura a seguir, na qual os botões estão todos colocados, mas seu funcionamento vai ser determinado por cada tipo de aparelho que o implementar.

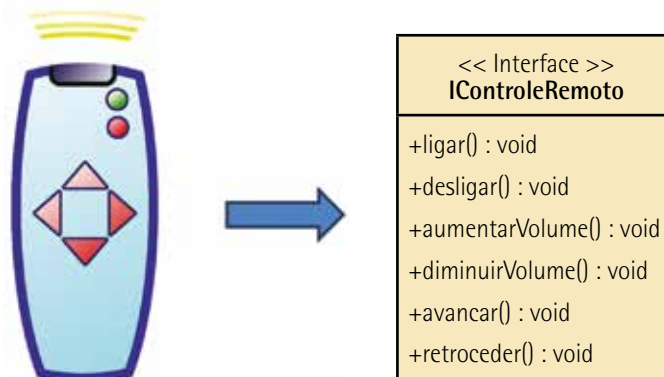


Figura 109 – Abstração de uma interface de controle remoto

Uma vez feito o UML, passamos a traduzir para o C#:

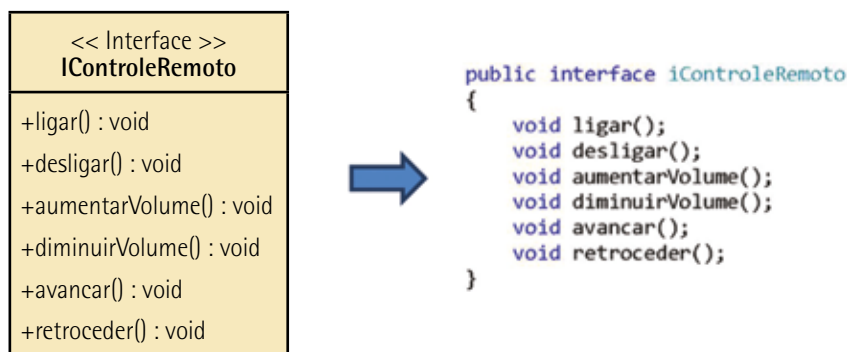


Figura 110 – Transformando o UML em código

Cada um dos métodos-guia da interface é uma abstração dos botões do objeto real controle remoto:

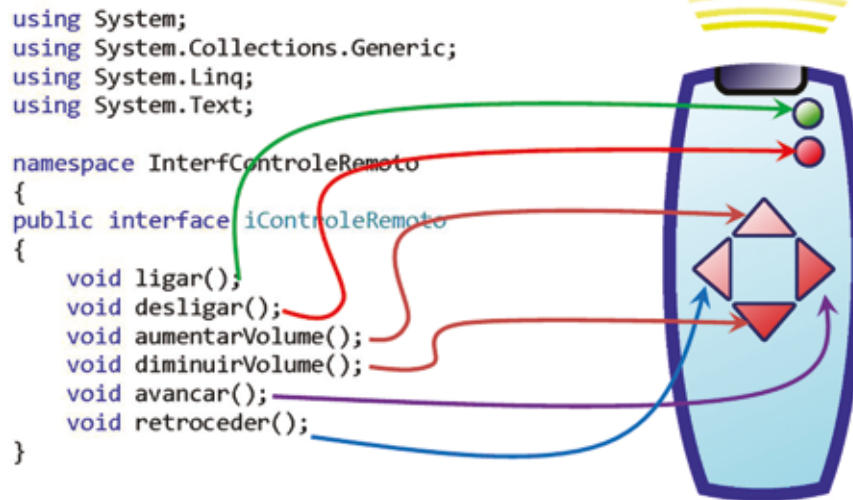


Figura 111 – Interface Controle Remoto

Como vimos, vamos implementar o controle remoto em três aparelhos. As ligações dessa interface com as classes são feitas como numa herança normal, mas a linha é tracejada, já que é uma realização:

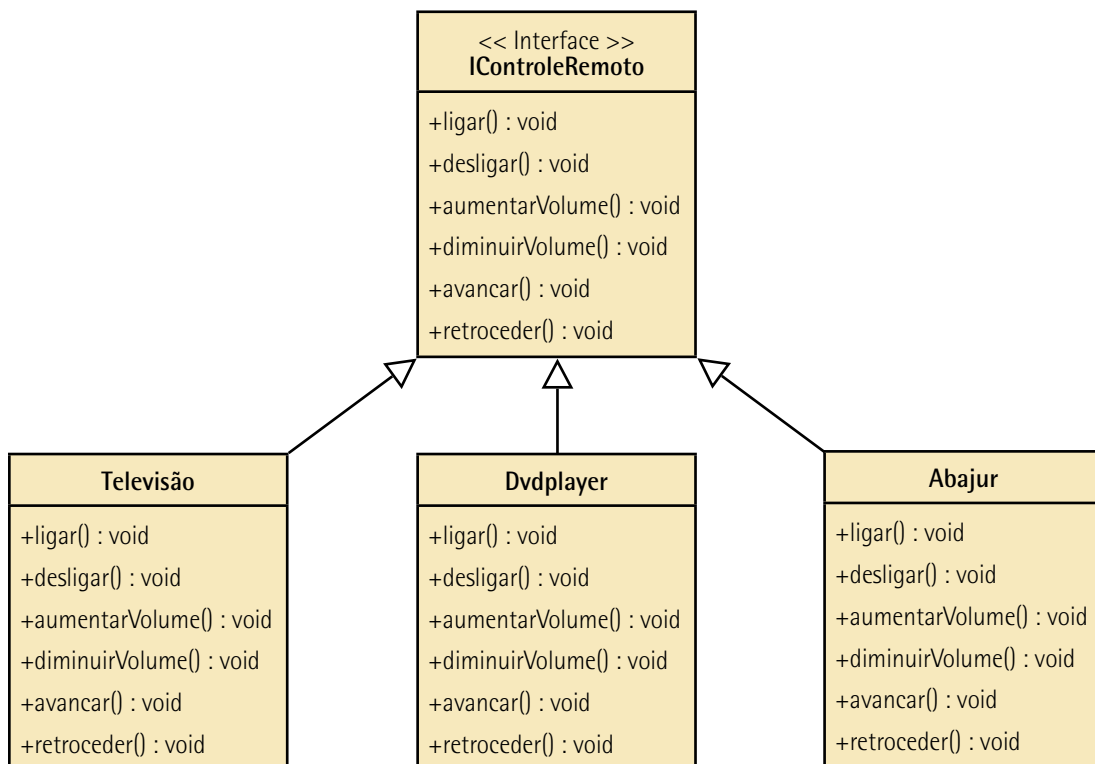


Figura 112 – UML das implementações do controle remoto

Uma vez montada a interface, vamos montar cada uma das classes que realizarão a sua implementação. Na classe Televisão, as teclas do avançar e retroceder irão mudar de canal:

```

public class Televisao : iControleRemoto
{
    private string marca;
    private string modelo;
    private bool ligada;
    private int nivelVolume;
    private int canal;

    public Televisao(string marca, string modelo)
    {
        this.marca = marca;
        this.modelo = modelo;
        ligada = false;
        canal = 0;
    }

    public virtual void ligar()
    {
        ligada = true;
    }

    public virtual void desligar()
    {
        ligada = false;
    }

    public virtual void aumentarVolume()
    {
        nivelVolume++;
    }

    public virtual void diminuirVolume()
    {
        nivelVolume--;
    }

    public virtual void avancar()
    {
        canal++;
    }

    public virtual void retroceder()
    {
        canal--;
    }

    public override string ToString()
    {
        string estado = ligada ? "ligada" : "desligada";
        return "Televisao: " + marca + " " + modelo + " " + estado;
    }
}

```

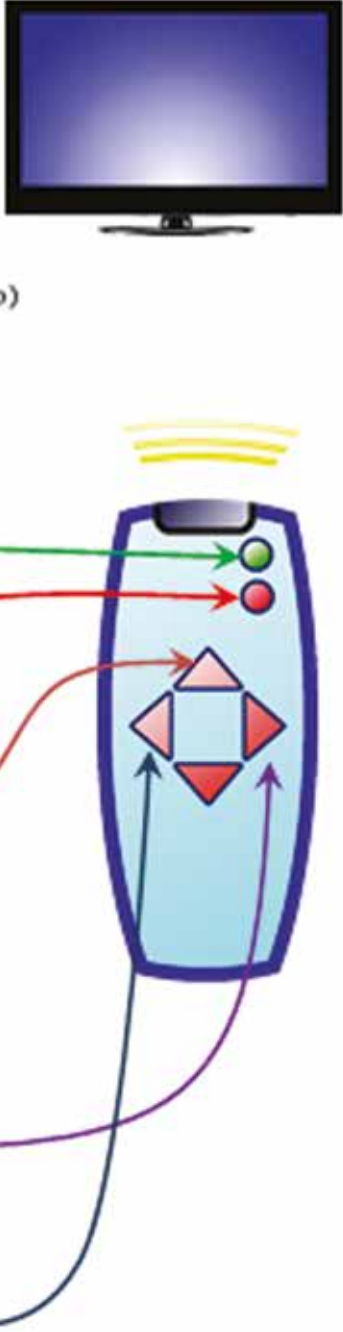


Figura 113 – Métodos implementados no controle remoto pela classe Televisão

No aparelho de DVD não temos volume, mas os botões de aumentar e diminuir volume serão utilizados para abrir e fechar a gaveta (combo), e o avançar e retroceder, para controlar os capítulos do filme:

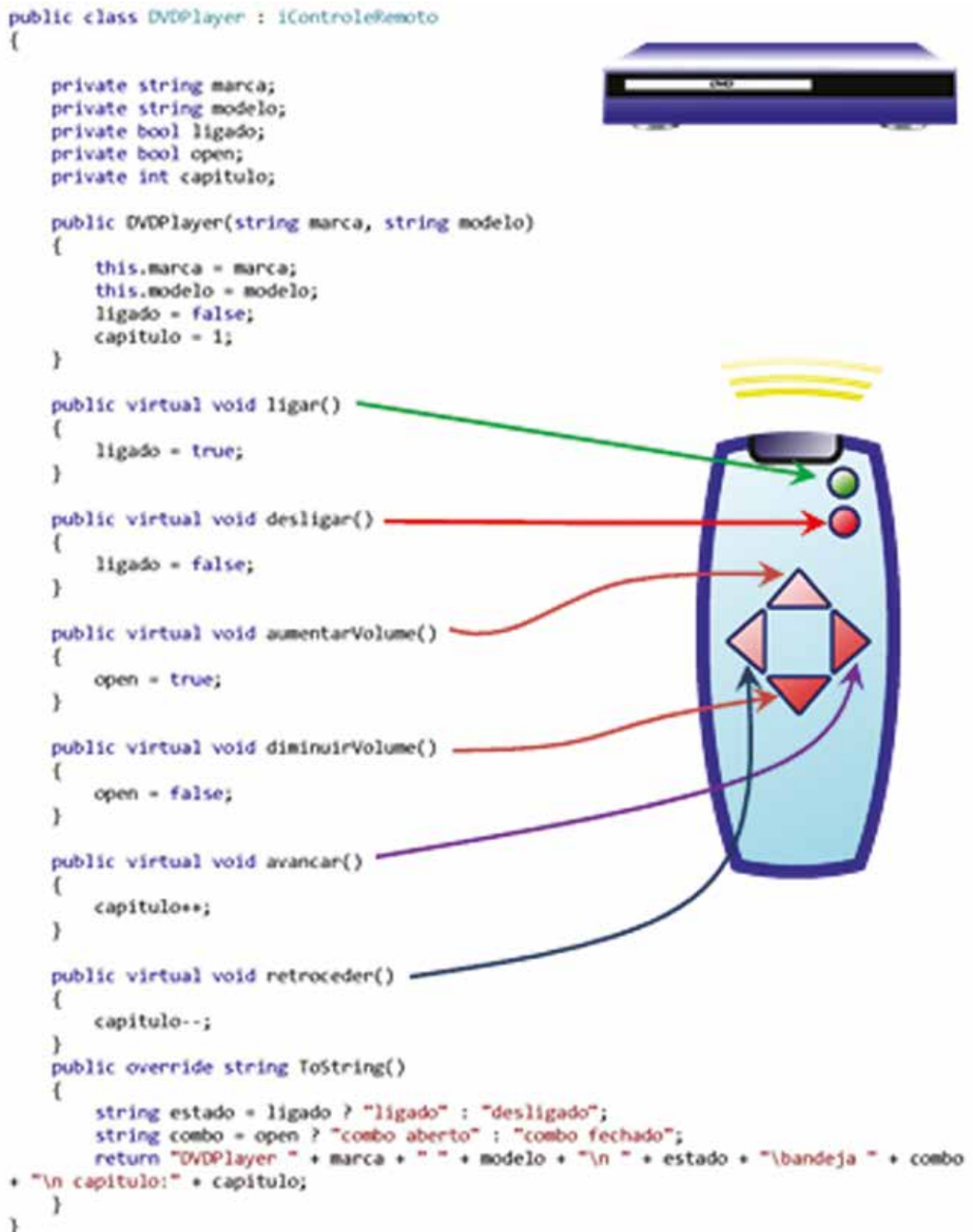


Figura 114 – Métodos implementados pelo DVD player no controle remoto

Finalmente, na classe Abajur, somente os botões de liga e desliga terão função. As outras teclas continuarão existindo, mas sem utilidade, porque o método existe mas não tem comandos (veja a figura a seguir). Note que os métodos estão todos presentes, pois a interface obriga a sua existência.

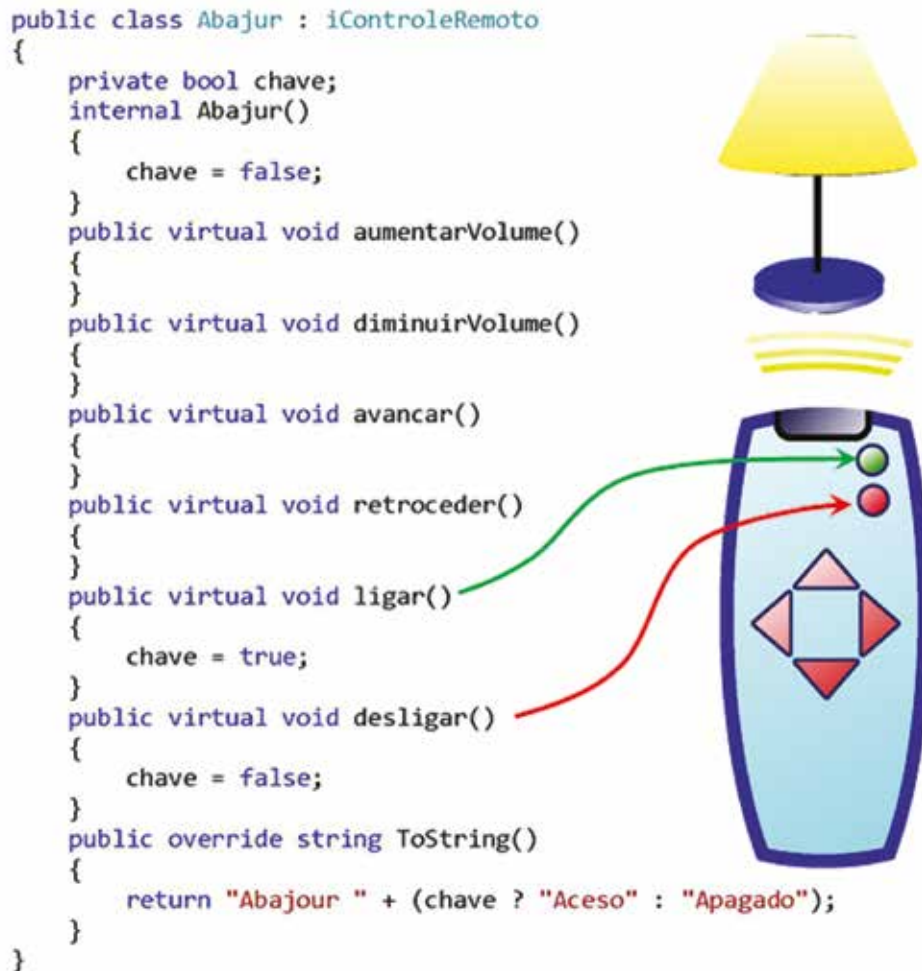


Figura 115 – Métodos implementados no controle remoto pela classe Abajur

Uma vez montadas as classes, vamos fazer o teste criando os seus objetos e verificando o funcionamento dos botões (veja o código a seguir):

Código 85 – Programa funcionando com o controle remoto

```
class Program
{
    static void Main(string[] args)
    {
        // ControleRemoto philco = new Televisao("Philco", "MultiVision");
        iControleRemoto philco = new Televisao("Philco", "MultiVision");
        iControleRemoto dvdGradiente = new DVDPlayer("Gradiente", "D22");

        philco.ligar();
        dvdGradiente.ligar();

        Console.WriteLine("Aparelhos iniciais:");
        Console.WriteLine(philco);
        Console.WriteLine(dvdGradiente);

        philco.desligar();
        ((DVDPlayer)dvdGradiente).aumentarVolume();

        Console.WriteLine("");
        Console.WriteLine("Aparelhos apos desligar a TV e abrir o combo do DVD:");
        Console.WriteLine(philco);
        Console.WriteLine(dvdGradiente);

        iControleRemoto alibaba = new Abajur();
        alibaba.ligar();
        Console.WriteLine(alibaba);
    }
}
```

Ele irá resultar na seguinte tela:

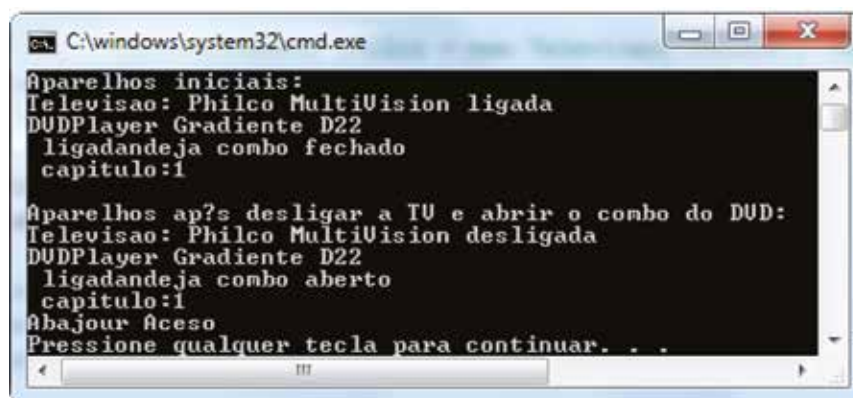


Figura 116 – Saída do teste do controle remoto

Exemplo de aplicação

Para praticar, experimente fazer outras operações como ligar e desligar a televisão, mudar de canal, mudar de capítulo no DVD, ligar e desligar estes aparelhos.



Lembrete

Para efeito de padronização, nomeamos os nomes das *interfaces* iniciando com a letra i maiúscula (I) o nome da interface. Isso ajuda os desenvolvedores a saberem que, se eles utilizarem aqueles métodos em suas classes, por exemplo, ele terão que implementar o código de processamento com base nas entradas e saídas predefinidas nas *interfaces*.

7.3 Herança simples e múltipla

Conforme já abordado, quando uma classe é uma derivação de outra já existente, temos uma relação de herança – nesse caso, uma herança simples. Há um segundo conceito, chamado herança múltipla, que significa que uma classe pode ser derivada de mais de uma classe já existente.

A princípio, a herança múltipla traz certas vantagens, pois teríamos uma classe derivada de diversas outras já existentes. Porém, o fato é que o controle desse tipo de estrutura é bem complexo e poucas linguagens de programação permitem realizar esse conceito – normalmente, elas permitem apenas a herança simples.

Por outro lado, a modelagem visando à herança múltipla pode trazer essa complexidade para a manutenção do sistema, gerando assim uma maior mão de obra no que se refere a atualizações e alterações no sistema.

Como não são todas as linguagens de programação que permitem herança múltipla (uma classe herdar características de mais de uma classe), a técnica de interface nos permite agregar em uma classe várias *interfaces* distintas.

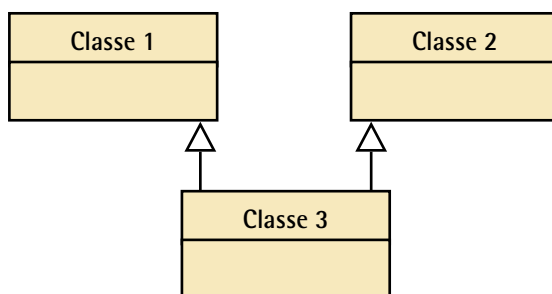


Figura 117 – Representação de uma herança múltipla

A herança múltipla pode causar um sério problema, o chamado diamante da morte. Ela acontece quando há uma herança múltipla de duas classes que são herdeiras de uma mesma classe, conforme visto na figura a seguir. As classes B e C herdarão o método1 da classe A. Por sua vez, a classe D, ao herdar de B e de C, receberá o método1 de ambos. Isso causa um problema, já que teremos uma duplicidade, e ocorrerá um erro. Algumas linguagens estabelecem uma hierarquia para contornar o problema.

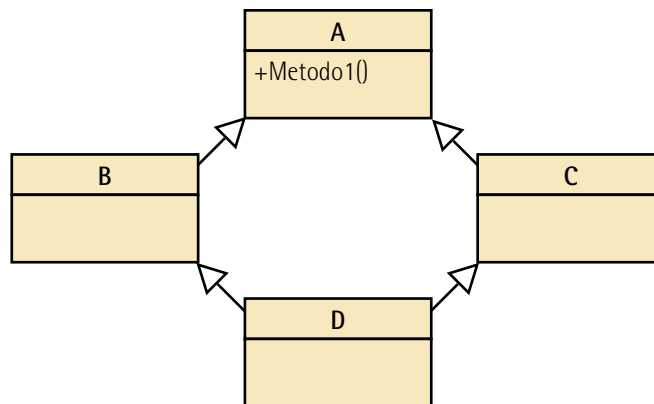


Figura 118 – UML diamante da morte

A linguagem C#, assim como o Java, não permite o uso de herança múltipla. A situação pode ser simulada com o uso de interface, mas ficando claro que não é uma herança múltipla dentro da sua definição.

Apresentando um exemplo simples, é uma classe que recebe duas *interfaces*, uma que determina características de um meio de transporte voador e outra que determina as características de um meio de transporte navegador. Veja o código:

Código 86 – Interface de um meio de transporte voador

```
public interface IVoador
{
    void Voar();
}
public interface INavegador
{
    void Navegar();
}
```

Aqui, um avião é um voador, conforme o código a seguir, e um transatlântico (que não está no exemplo), um navegador.

Código 87 – Implementação da interface Voador

```
public class Aviao : IVoador
{
    public void Voar()
    {
        Console.WriteLine("O avião voa");
    }
}
```

Agora, o caso de uma múltipla herança, um hidroavião:

Código 88 – Hidroavião implementado com as *interfaces* Voador e Navegador

```
public class HidroAviao : IVoador, INavegador
{
    public void Voar()
    {
        Console.WriteLine("O Hidroavião voa");
    }
    public void Navegar()
    {
        Console.WriteLine("O Hidroavião navega");
    }
}
```

Uma vez montadas as classes, vamos montar aquela que irá testar as classes. Ela irá criar uma instância de um avião chamado "boeing", que voará, e de um hidroavião chamado "catalina", que voará e navegará:

Código 89 – Criando um Boeing (avião) e um Catalina (hidroavião)

```
class Program
{
    static void Main(string[] args)
    {
        IVoador boeing = new Aviao();
        boeing.Voar();
        HidroAviao catalina = new HidroAviao();
        catalina.Voar();
        catalina.Navegar();
    }
}
```

Desta forma, a saída mostra o avião voando e o hidroavião voando e navegando.

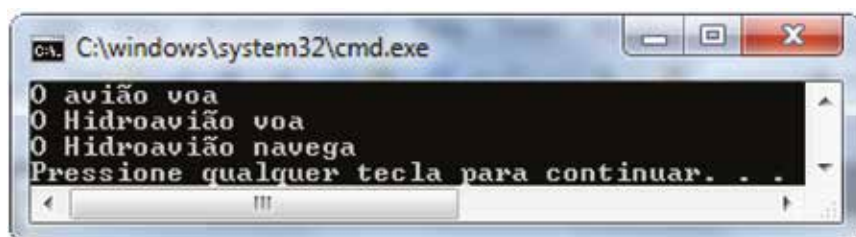


Figura 119 – Saída do programa com as instâncias do avião que voa e do hidroavião que voa e navega

7.4 Hierarquias de classes

Quando a modelagem de um sistema em POO chega à necessidade de derivação entre classes (herança), é altamente aconselhável que a classe Pai (classe principal) seja constituída da maior parte possível dos métodos e atributos que serão utilizados das derivações (classes Filhas). Esse cuidado previne redundância de informações, o que facilita a manutenção do sistema. A geração de um conjunto de derivações partidas de uma classe Pai tem o nome de hierarquia de classes.

Um projeto de um sistema OO começa sempre com a definição da hierarquia de classes descrevendo as relações dos objetos que irão formar o programa.

É comum, quando estudamos uma linguagem de programação orientada a objetos, entrarmos em contato a hierarquia de suas classes. Este estudo ajuda a entender o funcionamento da linguagem de programação; assim, conhecemos as classes já existentes e evitamos retrabalho.

Classes que declaram mas não implementam métodos são particularmente úteis na criação de hierarquias de classes porque não permitem a criação de instâncias delas e exigem que as classes descendentes implementem os métodos declarados nelas.

Neste momento é interessante estudar a classe **Object**. Ela suporta todas as classes na hierarquia de classes e fornece serviços de baixo nível para classes derivadas. A classe Object é classe base fundamental de todas as classes; ela é a raiz da hierarquia de tipos.

Como todas as classes são derivadas da Object, todo método definido nesta está disponível em todos os demais objetos no sistema. A classe Object é a superclasse de todas as classes, e todas são herdeiras dela.

No quadro a seguir temos os métodos da classe Object. Eles serão herdados nas classes derivadas que podem sobrescrever e sobrescrevem alguns desses métodos, já que elas são declaradas virtuais.

Quadro 19 – Métodos da classe Object

Nome	Descrição
<i>Equals</i> (Object)	Verifica se o objeto é igual ao objeto atual.
<i>Equals</i> (Object, Object)	Determina se as instâncias dos objetos são iguais.
<i>Finalize</i>	Permite um objeto tentar liberar recursos e executar outras operações de limpeza antes que ele seja recuperado pelo <i>garbage collector</i> .
<i>GetHashCode</i>	Serve como a função de <i>hash</i> padrão.
<i>GetType</i>	Obtém o Tipo da instância atual.
<i>MemberwiseClone</i>	Cria uma cópia do Object atual.
<i>ReferenceEquals</i>	Determina se as instâncias especificadas de Object é a mesma instância.
<i>ToString</i>	Retorna uma cadeia que representa o objeto atual.

Vamos construir então uma classe (Ponto) utilizando alguns métodos da classe Object. No método sobrescrito *Equals()*, ele recebe como parâmetro um objeto do tipo *object* e dentro dele é utilizado o método *GetType()* para obter o tipo da instância corrente. No método sobrescrito *GetHashCode()*, é feita uma simulação do cálculo de um *Hash* particular para caracterizar o objeto, e não o valor devolvido pelo método padrão. O último método sobrescrito, o *ToString*, já foi utilizado em alguns programas exemplos, mas sem explicação e esclarecendo agora. Uma *string* pode ser mostrada quando solicitamos para escrever na saída e simplesmente colocamos o nome da instância. Por padrão, o método *ToString* mostra o caminho da herança do objeto. A classe ainda tem um método *Copy*, que utiliza o método *memberWiseClone()*, e este cria uma nova instância do objeto corrente:

Código 90 – Classe ponto utilizando métodos da classe Object

```
class Ponto
{
    public int x, y;

    public Ponto(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override bool Equals(object obj)
    {
        if (obj.GetType() != this.GetType())
            return false;
        Ponto other = (Ponto)obj;
        return (this.x == other.x) && (this.y == other.y);
    }

    public override int GetHashCode()
    {
        return x ^ y;
    }
    public override String ToString()
    {
        return String.Format("({0}, {1})", x, y);
    }

    public Ponto Copy()
    {
        return (Ponto)this.MemberwiseClone();
    }
}
```

Agora, vamos testar os métodos herdados da classe `Object`. No primeiro teste, vamos criar um novo objeto utilizando o método `Ponto.Copy()` aplicando o `Object.MemberwiseCopy()` a partir da instância `p1` da classe `Ponto`. Uma vez criados os dois objetos, deve-se verificar se realmente são dois objetos ou apenas um mesmo objeto com duas referências diferentes apontando para ele. Fazemos esta operação utilizando o método `ReferenceEquals()`:

Código 91 – Duas instâncias da classe `Ponto` criadas

```
class Program
{
    static void Main(string[] args)
    {
        Ponto p1 = new Ponto(1, 2);
        Ponto p2 = p1.Copy();
        if (Object.ReferenceEquals(p1, p2))
            Console.WriteLine("As referências são iguais");
        else
            Console.WriteLine("As referências são diferentes");
    }
}
```

Ao executar, deve-se verificar se trata-se realmente de dois objetos independentes, já que o método `ReferenceEquals()` devolveu `False`:

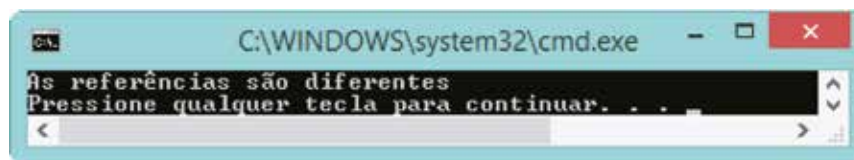


Figura 120 – Saída do método `ReferenceEquals()`

No mesmo programa, vamos verificar se os dois objetos são iguais utilizando o método sobrescrito `Equals()`:

Código 92 – Verificando se as duas instâncias são iguais

```
class Program
{
    static void Main(string[] args)
    {
        Ponto p1 = new Ponto(1, 2);
        Ponto p2 = p1.Copy();
        if (Object.Equals(p1, p2))
            Console.WriteLine("Os Objetos são iguais");
        else
            Console.WriteLine("Os Objetos são diferentes");
    }
}
```

A saída mostra que os dois objetos são iguais (veja a figura a seguir), já que o segundo objeto foi criado pelo método *MemberwiseClone()*.

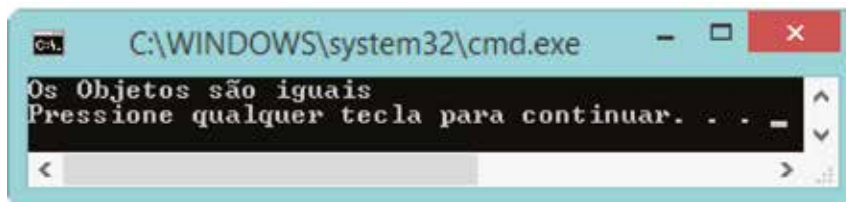


Figura 121 – Saída do programa usando o método Equals

Vamos alterar o programa, desta vez criando o objeto p1 e atribuindo a referência em p3:

Código 93 – Uma instância mas com duas referências apontando para ele

```
class Program
{
    static void Main(string[] args)
    {
        Ponto p1 = new Ponto(1, 2);
        Ponto p3 = p1;
        if (Object.ReferenceEquals(p1, p3))
            Console.WriteLine("As referências são iguais");
        else
            Console.WriteLine("As referências são diferentes");
    }
}
```

Vemos que as duas referências apontam para o mesmo objeto:

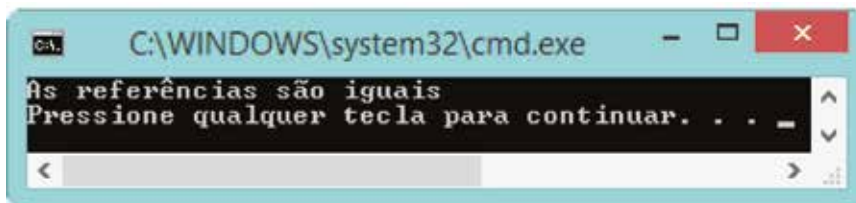


Figura 122 – Saída apresentada quando duas referências apontam para o mesmo objeto

Agora vamos verificar o valor do *Hash* utilizando o método *GetHashCode()* sobrescrito na classe *Ponto*:

Código 94 – Utilizando o *Hashcode* sobrescrito pela classe

```
class Program
{
    static void Main(string[] args)
    {
        Ponto p1 = new Ponto(1, 2);
        Console.WriteLine("O Hashcode p1 é: {0}", p1.GetHashCode());
    }
}
```

O programa devolve a operação XOR entre os valores de x e y, que foi definido no método sobrescrito:

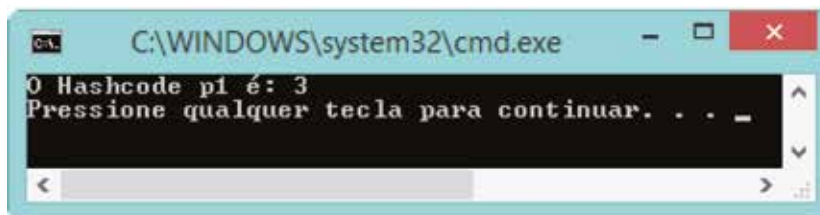


Figura 123 – Saída usando *GetHashCode* sobrescrito

Código 95 – Retirando o método *GetHashCode()* sobrescrito

Vamos alterar retirando o método sobrescrito da classe *Ponto*:

```
/*          public override int GetHashCode()
    {
        return x ^ y;
    }
*/
```

Dessa forma, podemos ver como a classe *Objeto* devolve o valor do *Hash* originalmente ():

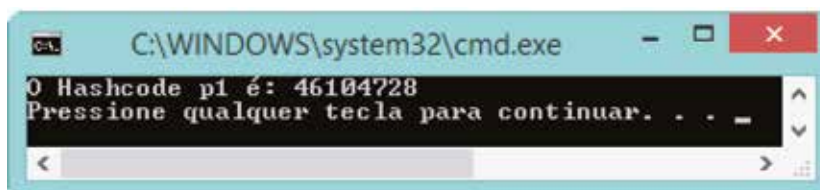


Figura 124 – Saída usando *GetHashCode* padrão

Para testar o método *ToString*, vamos inicialmente utilizar o método sobrescrito na classe *Ponto*, que devolve um texto definido pelo usuário. Assim, basta colocar no código o nome da referência da instância, sem colocar explicitamente o nome do método:

Código 96 – Utilizando o método *ToString* sobrescrito

```
class Program
{
    static void Main(string[] args)
    {
        Ponto p1 = new Ponto(1, 2);
        Console.WriteLine("Os valores de p1 são: {0}", p1);
    }
}
```

A saída mostra o conteúdo passado como parâmetro:

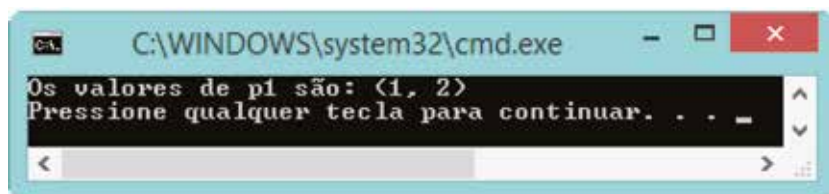


Figura 125 – Saída do método *ToString* sobrescrito

Retirando o método sobrescrito, podemos ver como é a saída do método *ToString()* original:

Código 97 – Retirando a sobreescritão do Método *ToString*

```
/*      public override String ToString()
        {
            return String.Format("{0}, {1})", x, y);
        }
*/
```

Assim, o método *ToString()* mostra o nome da classe:

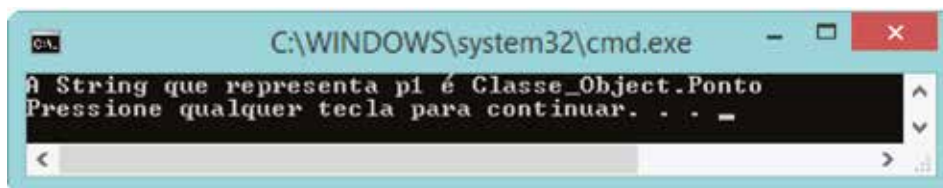


Figura 126 – Saída usando o método *ToString* da classe *Object*

```
class Program
{
    static void Main(string[] args)
    {
        Ponto p1 = new Ponto(1, 2);
        Console.WriteLine("A String que representa p1 é {0}", p1);
    }
}
```

Ao projetarmos um sistema OO, para que a hierarquia esteja sempre íntegra, devemos fazer as seguintes considerações:

- Todo objeto é instância de uma e só uma classe. Um objeto não pode ser instanciado a partir de duas classes.
- Em todos os objetos de uma mesma classe:
 - Cada um tem um conjunto de atributos de informação, que determinam o "estado" do objeto a cada momento, atributos estes que são as variáveis de instância ou campos, independentes entre si, desde que não estáticos.
 - Cada um tem um conjunto de funcionalidades – "ações" ou "serviços" que podem ser solicitados por meio de "mensagens" enviadas pelos métodos.
 - Duas classes diferentes precisam ser distintas: ter algum atributo, ou algum método, ou ambos, diferentes. Caso contrário, não teria sentido nomeá-las de forma diferente se são iguais.
- Duas classes que são subclasses de uma superclasse herdam os atributos e métodos comuns desta última, mas possuem outros atributos e métodos que as particularizam.
- Em muitas linguagens orientadas a objetos, as classes formam uma hierarquia simples, ou seja, há uma raiz de todas as classes, que é a classe Object. Todas as demais classes têm necessariamente uma única superclasse, mas podem ter ou não várias subclasses. Essa estrutura é a de uma árvore invertida.
- Existe uma relação "é-um" entre um objeto e sua classe, que "é um" para a superclasse desta e propaga para todas as superclasses acima na hierarquia até a classe Object (por exemplo: um tigre "é um" mamífero, que "é um" animal que "é um" objeto).
- Todo objeto é instância de uma classe. Um objeto obedece, por meio de uma referência, o comportamento da classe.
- Podemos redefinir, em uma subclasse, um método com a mesma assinatura de outro já existente na sua superclasse ou mais acima na hierarquia. No caso, esse novo método passará a substituir o

método da superclasse. Esse mecanismo chama-se *overriding*, ou sobrescrita, e é uma das formas de polimorfismo.

- Um atributo de uma classe pode referenciar qualquer instância dessa classe e qualquer instância das suas subclasses, mas não pode referenciar objetos de sua superclasse ou de outras classes acima na hierarquia.
- Quando um objeto recebe uma mensagem, inicialmente procura um método com a mesma assinatura. Se encontrar, executa esse método. Caso não encontre, passa a procurar na sua superclasse, na superclasse dela e assim por diante, executando o primeiro método que encontrar com a mesma assinatura. Caso alcance a classe Object e nem lá encontre esse método, ocorrerá um erro de execução.

8 RECURSOS DOS PROGRAMAS ORIENTADOS A OBJETOS

8.1 Persistência

Uma vez que são definidas as classes e que elas são utilizadas dentro do programa em forma de objetos, estes podem assumir duas formas de existência dentro do programa: objetos transientes ou objetos persistentes.

Objetos transientes são dados que trabalham em meios voláteis, ou seja, em memória. Uma vez finalizada a execução do programa, todos os dados referentes ao objeto serão perdidos, não podendo ser recuperados.

Objetos persistentes são dados que trabalham em meios não voláteis, ou seja, seus dados são armazenados em disco a partir de arquivos ou banco de dados. Isso permite que, uma vez finalizada a execução do programa, os dados do objeto possam ser recuperados na próxima execução.

Essa técnica também pode ser aplicada para transferência de dados quando se deseja transferir os dados de um objeto para outros programas de maneira fidedigna. Tal transferência é útil para a comunicação entre aplicativos e para a replicação de dados, especialmente em casos de replicação de dados assíncrona.

Dessa forma, quando iniciamos o aprendizado de POO, é muito comum não utilizarmos persistência, deixando essa técnica para estudos mais avançados.



Saiba mais

O LINQ proporciona padrões intuitivos para o desenvolvimento de consultas e atualização de dados, além de poder ser estendida para dar suporte a qualquer tipo de banco de dados. O Visual Studio inclui rotinas de LINQ ou DLINQ que permitem o uso em coleções do .NET Framework, bancos de dados do SQL Server, conjuntos de dados ADO.NET e documentos XML.

Para maiores detalhes, consulte o capítulo 25, "Consultando informações em Bancos de dados", da seguinte obra:

SHARP, J. *Microsoft visual C# 2008 passo a passo*. Porto Alegre: Bookman, 2008.

8.2 Inicialização e destruição de objetos

Quando utilizamos nossas classes em nossos programas, como discutido anteriormente, nós trabalhamos com objetos. Portanto, torna-se importante a compreensão sobre o tempo de vida de um objeto dentro do programa.

É comum, até certo ponto, quando iniciamos os estudos de POO e construímos nossos primeiros programas, ignorarmos completamente esses conceitos e seguirmos a diante. Porém, é importante saber que a compreensão sobre o tempo de vida de um objeto está ligada a como a linguagem de programação lida com esses objetos em memória.

As linguagens de POO mais antigas costumam acumular os objetos criados, deixando a cargo do programador realizar a destruição deles quando não utilizar mais. Nesse caso, vale salientar que não adianta finalizar a execução do programa principal, pois os objetos instanciados dentro do programa e que não foram destruídos no momento oportuno continuam na memória. Assim sendo, a única maneira de limpar a memória desses objetos é desligando o computador.

Já as linguagens de POO mais recentes possuem ferramentas que possibilitam que, uma vez que o objeto instanciado fique sem utilização (sem referências), tais ferramentas se encarreguem de limpar a memória desses resíduos. Não por acaso, essas ferramentas possuem nomes sugestivos como "coletores de lixo".

Embora essas linguagens realizem o trabalho de destruição de objetos que não estejam mais sendo utilizados, é muito salutar que o desenvolvedor crie o hábito de se preocupar, pelo menos no momento do desenvolvimento da classe, em como será a destruição dela.

Assim como toda classe possui um método especial chamado construtor, cujo objetivo é definir processos que ocorrerão unicamente quando o objeto for instanciado na memória, existe também um método especial chamado destruidor (ou desconstrutor). Esses métodos destruidores têm o objetivo de definir processos que serão executados apenas quando o objeto for removido da memória, ou seja, destruído.

Esses métodos tornam-se muito importantes à medida que a complexidade do seu projeto aumenta, pois podemos utilizar os processos desses métodos para liberação ou alocação de recursos em dispositivos (impressora, porta serial etc.), controle de arquivos, conexão e desconexão ao banco de dados etc.

Portanto, o tempo de vida de um objeto abrange desde sua instância em memória até sua liberação, passando pelos processos de criação e destruição, que devem ser definidos na elaboração da classe.

8.3 Tratamento de exceções

Em qualquer sistema, OO ou não, não basta que ele apenas esteja com a programação benfeita. Ele também deverá estar preparado para tratar de problemas (exceções) que venham a ocorrer. Nenhum sistema está imune às exceções e em algum momento qualquer sistema deverá estar preparado para tratar de um problema. Uma exceção em tempo de execução poderia encerrar o sistema de forma abrupta, mostrando uma mensagem incompreensível, levando o usuário a ficar sem ação ou mesmo danificando os dados que estavam sendo processados.

Para tratar esses problemas, é importante conhecer algumas técnicas de controle de erros, chamada de controle de exceção.

A linguagem C# adota um estilo relativamente comum em outras linguagens, que é o de tratar erros como objetos que encapsulam todas as informações que necessitamos para resolvê-lo. Essa ideia é válida para todo o ambiente .NET e foi batizada como SEH (Structured Exception Handling), conforme Lima e Reis (2002). O formato básico para tratamento de uma exceção é o seguinte:

Código 98 – Estrutura de captura e tratamento de erros

```
try
{
    // Código sujeito a exceções
}
catch (TipoExcecao1 e)
{
    // Tratamento para exceção tipo 1
}
catch (TipoExcecao2 e)
{
    // Tratamento para exceção tipo 2
}
catch
{
    // Tratamento para qualquer tipo de exceção
}
finally
{
    // Trecho que deve sempre ser executado, havendo ou não exceção
}
```

A estrutura *try..Catch* requer que exista pelo menos um bloco *Catch* vazio. Os blocos de *Finally* e *Catch*(com exception) são opcionais. Os comandos de tratamento de exceções podem ser vistos no quadro a seguir:

Quadro 20 – Comandos de tratamento de exceções

Comando	Descrição
<i>try</i>	Intervalo do código onde pode ocorrer o erro.
<i>Catch</i>	Intervalo de código onde o erro será tratado.
<i>Finally</i>	Essa instrução é executada sempre, utiliza-se para liberar algum recurso independente da ocorrência do erro.
<i>Throw</i>	Comando que gera exceções específicas.

Consideremos um exemplo simples. No código a seguir, temos um programa que, ao digitar no editor, não acusa nenhum erro:

Código 99 – Programa sem erro no editor

```
class Program
{
    static void Main(string[] args)
    {
        String v = "ABC";
        double num = Convert.ToDouble(v);
        Console.WriteLine(num);
    }
}
```

Ao executar o programa, porém, ele é interrompido, mostrando uma grande mensagem de erro:

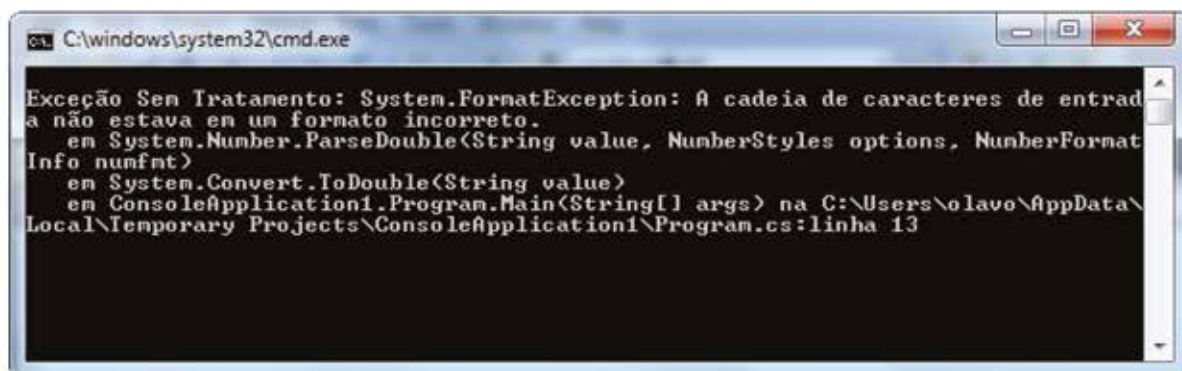


Figura 127 – Saída do programa do código anterior

O erro acontece porque a função *Convert.ToDouble()*, que transforma uma *string* em um número do tipo *double*, está tentando transformar uma *string* que não contém um número ("ABC"). Esse tipo de erro ocorre muito quando é feita a leitura do teclado de um número e a função *Console.ReadLine()*

retorna uma *string* que necessita ser convertida. O usuário poderia digitar uma palavra em vez de digitar um número. A maneira de corrigir é fazer um tratamento desse erro.

O trecho problemático deve ser envolvido pelo bloco do *try*. No bloco do *Catch*, é feito o tratamento do erro, conforme o código a seguir. Note que logo após a palavra *Catch* é feita a classificação do tipo de erro; dessa forma, o tratamento do erro é específico para um tipo de erro (no caso *FormatException*). Um *try* pode ter vários *Catch*, cada um deles tratando um tipo de erro. O C# contém uma série de classes para manipulação de exceções e é uma boa prática utilizarmos essas classes, conforme o quadro a seguir.

Código 100 0 – Tratando um erro de formato numérico

```
class Program
{
    static void Main(string[] args)
    {
        String v = "ABC";
        try
        {
            double num = Convert.ToDouble(v);
            Console.WriteLine(num);
        }
        catch (System.FormatException err)
        {
            Console.WriteLine("Erro de formato, mensagem: " + err.Message);
        }
    }
}
```

Quadro 21 – Algumas classes para o tratamento de erros

Exception	Descrição
<i>DivideByZeroException</i>	Erro gerado quando dividimos números inteiros por zero.
<i>IndexOutOfRangeException</i>	Erro gerado quando acessamos posições inexistentes de um <i>array</i> .
<i>FormatException</i>	Erro gerado quando acontece um problema na conversão.
<i>NullReferenceException</i>	Erro gerado quando utilizamos referências nulas.
<i>InvalidCastException</i>	Erro gerado quando realizamos um <i>casting</i> incompatível.

Ao executar o programa, não causa abandono, prosseguindo a execução após o tratamento:

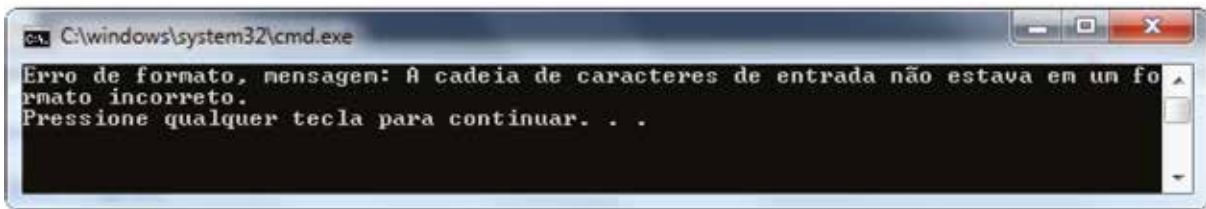


Figura 128 – Saída do problema tratado



Lembrete

Fazer um bloco *try* muito grande para evitar um erro pode causar problemas, pois pode-se perder a referência exata do local do erro e o programa pode prosseguir sem ter o erro tratado corretamente. O ideal é tratar um erro por vez.

Existem ocasiões nas quais podemos utilizar o tratamento de erros como recurso de programação, e não somente para erros de lógica. Nessas ocasiões, podemos fazer o programa gerar propositalmente o próprio erro. Quando precisamos gerar as nossas próprias exceções, utilizamos a instrução *Throw*, que geralmente é utilizada quando queremos fazer alguma validação.

No exemplo do código a seguir, vamos provocar um erro da classe *ArgumentException* quando em um método que calcula o aumento de salário é passado um número negativo. Normalmente, a ocorrência de um erro neste tipo de rotina é raro, mas é interessante provocar um erro quando não queremos que uma situação particular ocorra.

Código 101 – Provocando um erro

```
class Funcionario
{
    public void AumentaSalario(double aumento)
    {
        if (aumento < 0)
        {
            ArgumentException erro = new ArgumentException();
            throw erro;
        }
    }
}
```

No programa que utiliza o método para o aumento de salário, é passado um valor negativo, conforme o código a seguir:

Código 102 – Programa sem o tratamento do erro

```
class Program
{
    static void Main(string[] args)
    {
        Funcionario f = new Funcionario();
        f.AumentaSalario(-1000);
    }
}
```

Porém, como não é feito o tratamento do erro, acontece o abandono do programa, como ilustra a figura:

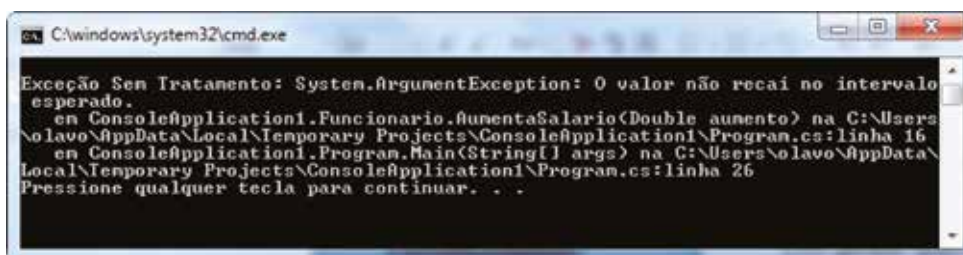


Figura 129 – Saída do erro provocado sem tratamento

Para fazer o tratamento, colocamos a estrutura do *try...Catch* envolvendo o trecho em que há a necessidade verificar se a passagem de parâmetro está ocorrendo sem problema:

Código 103 – Tratando um erro gerado

```
class Program
{
    static void Main(string[] args)
    {
        Funcionario f = new Funcionario();
        try
        {
            f.AumentaSalario(-1000);
        }
        catch (ArgumentException e)
        {
            System.Console.WriteLine("Houve uma ArgumentException ao aumentar o salário");
        }
    }
}
```

No caso, acontece o erro e é feito o tratamento para um caso particular, mostrando o erro da figura a seguir sem que aconteça o abandono do programa.

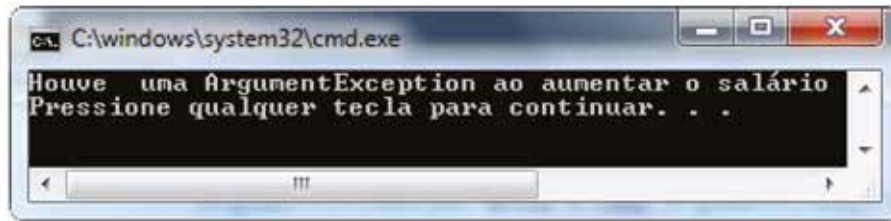


Figura 130 – Saída do programa com o erro provocado e tratado



Saiba mais

O MIT, que lançou alguns dos conceitos de orientação a objeto, disponibiliza um programa totalmente gráfico, com objetos que permitem criar de maneira simples aplicativos para o sistema Android, o Appinventor. Veja no site:

<<http://appinventor.mit.edu/>>.



Resumo

Nesta unidade, vimos como podemos combinar e integrar as classes a partir de diversas técnicas, mostrando um pouco de como é importante projetar bem a classe, pois seus aspectos de funcionamento influenciam diretamente no desempenho e na estrutura do programa. Assim, toda a classe pode ser combinada com características de outras classes, sendo que tais combinações podem ser feitas por associações, agregações ou heranças. Existem dois tipos de heranças: as simples e as múltiplas. Herança simples ocorre quando uma classe herda as características de uma única classe. Já a herança múltipla se dá em casos que uma classe herda as características de mais de uma classe. Sobre isso, estudamos que nem todas as linguagens de programação permitem herança múltipla.

Vimos posteriormente que os objetos ficam instanciados em memória, porém, caso haja necessidade, podemos realizar a persistência de objeto, ou seja, gravá-lo em meios não voláteis (como um arquivo em disco) para que possa ser acessado em outro momento.

Outros dois conceitos importantes foram vistos, a classe abstrata e a interface. Quando definimos uma classe, usamos classes abstratas ou

interfaces que foram escritas com o objetivo de criar e definir padrões de funcionamento para classes que herdarão tais funcionalidades. As classes abstratas podem possuir métodos completos que serão herdados pela subclasse, além de métodos abstratos que serão implementados na subclasse. Interfaces são métodos com entradas e saídas predefinidas que servem para padronizar funcionamentos. Porém, a implementação de todos esses métodos ficam sendo de responsabilidade da classe que herdará a interface.

A destruição de objetos, ou seja, retirar o objeto da memória, também deve ser uma preocupação na definição das classes pelo desenvolvedor, pois, uma vez alocado o objeto na memória, deve ser liberado dela quando não for mais utilizado.

Finalmente vimos a técnica de tratamento das exceções, em que erros que não são de lógica podem ser tratados evitando a saída abrupta do programa e enviando mensagem ao usuário para providências a serem tomadas para sanar o problema. Para isso são utilizados os blocos do *try*, *catch* e *finally*.



Exercícios

Questão 1. (TCE/PA 2012) Em C#, os métodos chamados pelo mecanismo de execução do programa quando o objeto está prestes a ser removido da memória são denominados de:

- A) Garbage collection.
- B) Destruidores.
- C) Propriedades.
- D) Indexadores.
- E) Eventos.

Resposta correta: alternativa B.

Análise das alternativas

- A) Alternativa incorreta.

Justificativa: garbage collection é o mecanismo que algumas linguagens orientadas a objetos possuem para remover da memória objetos que não estão sendo usados/referenciados, ou seja, não são esses tipos de métodos.

B) Alternativa correta.

Justificativa: métodos destruidores são os métodos que as classes podem possuir para realizar alguma ação antes de disponibilizar os objetos para o mecanismo garbage collection retirá-lo da memória. Pode-se traçar um paralelo com o método construtor. Para instanciar/criar um objeto temos o método construtor. Antes de "destruir"/acabar com a referência de um objeto, temos o método destruidor.

C) Alternativa incorreta.

Justificativa: propriedades não são métodos. Em OO temos como propriedades de uma classe os seus atributos e métodos.

D) Alternativa incorreta.

Justificativa: indexadores não são métodos. Em OO temos como indexadores apenas os índices que estabelecemos para acessar as posições em estruturas como vetores e matrizes.

E) Alternativa incorreta.

Justificativa: eventos não são métodos. Existem um outro paradigma de programação assim como OO e estruturada chamado de orientado a eventos, que possui outros conceitos e características específicas.

Questão 2. (Dataprev 2011) Sobre a programação orientada a objetos, é incorreto afirmar que:

A) uma interface é uma classe abstrata que não pode ser instanciada.

B) as interfaces permitem explorar o polimorfismo.

C) uma classe abstrata significa que ela deve ser estendida; um método abstrato significa que ele deve ser sobreposto.

D) um método abstrato pode estar presente em uma classe não abstrata, desde que não tenha corpo.

E) é possível combinar métodos abstratos e não abstratos em uma classe abstrata.

Resolução desta questão na plataforma.

REFERÊNCIAS

Textuais

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML: guia do usuário*. 2. ed. Rio de Janeiro: Campus, 2006.

CAMARA, F. *Orientação a objeto com .NET*. 2.ed. Florianópolis: Visual Books, 2006.

CLASSE List<T>. [s.d.]. Disponível em: <[msdn.microsoft.com/pt-br/library/6sh2ey19\(v=vs.110\).aspx](http://msdn.microsoft.com/pt-br/library/6sh2ey19(v=vs.110).aspx)>. Acesso em: 27 out. 2014.

DEITEL, H. M. *Java, como programar*. Porto Alegre: Bookman, 2003.

FORMATAÇÃO composta. [s.d.]. Disponível em: <[msdn.microsoft.com/pt-br/library/txafckwd\(v=vs.110\).aspx](http://msdn.microsoft.com/pt-br/library/txafckwd(v=vs.110).aspx)>. Acesso em: 27 out. 2014.

FOWLER, M. *UML essencial: um breve guia para a linguagem-padrão de modelagem de objetos*. São Paulo: Bookman, 2006.

GUEDES, G. T. *UML: uma abordagem prática*. São Paulo: Novatec, 2006.

HERANÇA (guia de programação do C#). [s.d.]. Disponível em: <[http://msdn.microsoft.com/pt-br/library/vstudio/ms173149\(v=vs.110\).aspx](http://msdn.microsoft.com/pt-br/library/vstudio/ms173149(v=vs.110).aspx)>. Acesso em: 27 out. 2014.

LIMA, E.; REIS, E. *C# e .Net para desenvolvedores*. Rio de Janeiro: Campus, 2002.

LINDEN, R. *Algoritmos genéticos*. 2. ed. Rio de Janeiro: Brasport, 2008.

LINQ (Consulta Integrada à Linguagem). [s.d.]. Disponível em: <<http://msdn.microsoft.com/pt-br/library/bb397926.aspx>>. Acesso em: 27 out. 2014.

MANZANO, J. A. *Estudo dirigido de Microsoft Visual C# 2010 Express*. São Paulo: Érica, 2010.

SANTOS, L. C. *Microsoft visual C# 2008 Express Edition: aprenda na prática*. São Paulo: Érica, 2010.

SANTOS, R. *Introdução a programação orientada a objetos usando Java*. Rio de Janeiro: Campus, 2003.

SEBESTA, R. W. *Conceitos de linguagens de programação*. Porto Alegre: Bookman, 2011.

SHARP, J. *Microsoft visual C# 2008 passo a passo*. Porto Alegre: Bookman, 2008.

SINTES, T. *Aprenda orientação a objetos em 21 dias*. São Paulo: Pearson, 2002.

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

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

Site

<<http://appinventor.mit.edu/>>.

<<http://www.nr.no/en/about-main>>

<<http://www.visual-paradigm.com/download/community.jsp>>

<http://www.visualstudio.com/pt-br/downloads/download-visual-studio-vs#DownloadFamilies_2>

Exercícios

Unidade I – Questão 1: INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). *Exame Nacional de Desempenho dos Estudantes (ENADE) 2005*: Computação. Questão 20. Disponível em: <<http://download.inep.gov.br/download/enade/2005/provas/COMPUTACAO.pdf>>. Acesso em: 3 dez. 2018.

Unidade I – Questão 2: DPE/SP. *Agente de defensoria 2010*: programador. Questão 52. Disponível em: <<https://s3.amazonaws.com/files-s3.iesde.com.br/resolucaoq/prova/prova/24562.pdf>>. Acesso em: 3 dez. 2018.

Unidade II – Questão 1: TRF. *Analista do judiciário 2014*: informática. Questão 43. Disponível em: <<https://s3.amazonaws.com/files-s3.iesde.com.br/resolucaoq/prova/prova/38724.pdf>>. Acesso em: 3 dez. 2018.

Unidade III – Questão 1: TRT/SC. *Analista do judiciário 2013*: TI. Questão 31. Disponível em: <https://arquivos.qconcursos.com/prova/arquivo_prova/31432/fcc-2013-trt-12-regiao-sc-analista-judiciario-tecnologia-da-informacao-prova.pdf?_ga=2.40228578.1192330494.1543860174-787814988.1537967909>. Acesso em: 3 dez. 2018.

Unidade III – Questão 2: DPE/RJ. *Técnico superior especializado 2014*: analista de desenvolvimento de sistemas. Questão 71. Disponível em: <<https://s3.amazonaws.com/files-s3.iesde.com.br/resolucaoq/prova/prova/38121.pdf>>. Acesso em: 3 dez. 2018.

Unidade IV – Questão 1: TCE/PA. *Assessor técnico 2012*: analista de sistemas. Questão 94. Disponível em: <https://arquivos.qconcursos.com/prova/arquivo_prova/28820/aocp-2012-tce-pa-assessor-tecnico-de-informatica-analista-de-sistemas-prova.pdf?_ga=2.139424371.1192330494.1543860174-787814988.1537967909>. Acesso em: 3 dez. 2018.

Unidade IV – Questão 2: DATAPREV. *Analista de tecnologia da informação 2011*: desenvolvimento de sistemas. Questão 46. Disponível em: <<https://s3.amazonaws.com/files-s3.iesde.com.br/resolucaoq/prova/prova/28559.pdf>>. Acesso em: 3 dez. 2018.



Handwriting practice lines consisting of 30 horizontal blue lines. Each line is preceded by a small blue dot, serving as a starting point for letter formation. The lines are evenly spaced and extend across the width of the page.



Interativa

Informações:
www.sepi.unip.br ou 0800 010 9000