Programação Orientada a Objetos I

Notas de Aula 02

Prof. Alexandre Mignon

Observação

Os exemplos desse conjunto de slides estão no endereço:

https://sites.google.com/site/amignon/

Herança

- ➔ É um princípio da POO que permite que as classes compartilhem atributos e operações.
- ➔ É o mecanismo que permite construir novas classes a partir de classes existentes, facilitando o reuso de métodos e dados de uma classe em outra.
- ➔ A herança permite a criação de subclasses que herdam atributos e operações (ou métodos) da classe pai (superclasse ou classe base).

Herança

- A herança permite que você baseie a definição de uma nova classe em uma classe previamente existente. Quando você baseia uma classe em outra, a definição da nova classe herda automaticamente todos os atributos, comportamentos e implementações presentes na classe previamente existente.
- Instâncias da classe mais especializada (subclasse) podem acessar dados e comportamento (métodos) associados com a classe mais geral (ou superclasse).

Herança

- Herança é freqüentemente referida como o relacionamento "*é um*" entre duas classes.
- Os atributos e métodos de uma superclasse são herdados por suas subclasses.
  - Na especificação das subclasses pode-se adicionar novos atributos, novos métodos, e alterar ou sobrescrever métodos da superclasse.
- Herança não é cópia, é uma derivação.
  - Alterações realizadas na classe pai poderá afetar a classe filha, porém a classe filha poderá ser definida com comportamentos distintos da classe pai.

Herança

- ➔ Uma subclasse não tem acesso aos atributos e métodos privativos de sua superclasse.
- ➔ Caso se queira ter acesso a atributos e métodos da superclasse, pode-se utilizar o modificador de acesso ***protected***.

Tipos de Herança

➔ Existem três maneiras principais de usar herança:

- Para reutilização de implementação
- Para diferença
- Para substituição de tipo

Herança para Implementação

- ➔ Herança permite que uma nova classe reutilize implementação de outra classe.
- ➔ Evita o recortar/colar de código.
- ➔ Evita instanciar e usar um componente através de composição.
- ➔ A classe filha já nasce com todas as funcionalidades da classe pai.

Herança para Diferença

- ➔ Programação para diferença significa herdar uma classe e adicionar apenas o código que torne a nova classe diferente da classe herdada.
- ➔ Especialização é o processo de uma subclasse ser projetada em termos de como ela é diferente de sua superclasse. Isto é, a definição da subclasse inclui apenas os elementos que a tornam diferentes da superclasse.

Herança para Substituição de Tipo

- ➔ A substituição de tipos permite que você descreva relacionamentos com capacidade de substituição.
- ➔ Um relacionamento com capacidade de substituição significa que você pode usar qualquer instância de subclasse onde uma instância da superclasse é esperada.

Benefícios da Herança

- ➔ Reusabilidade de software
- ➔ Compartilhamento de código
- ➔ Consistência de interface
- ➔ Componentes de software
- ➔ Prototipação Rápida
- ➔ Polimorfismo e arcabouços

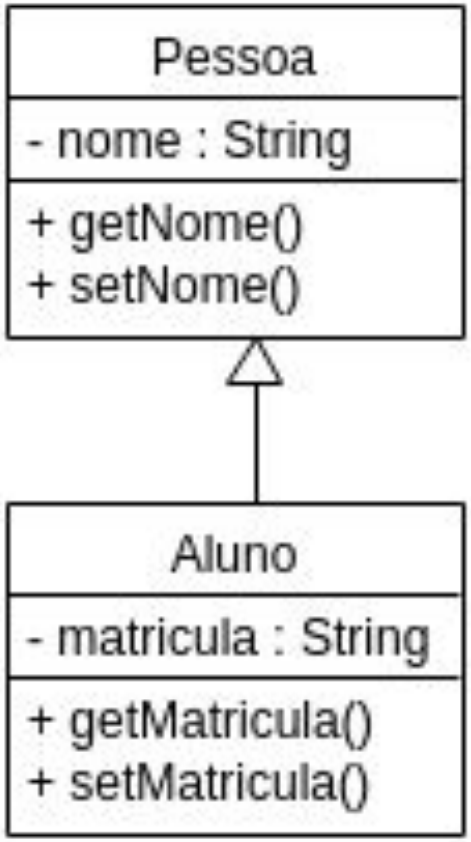
Herança em C#

➔ Em C# a identificação de herança acontece colocando dois pontos (:) e o nome da superclasse logo após a declaração da classe que será a subclasse.

```
class Pessoa {...}

class Aluno : Pessoa {...}
```

Exemplo de Herança I



Herança e Construtor

- ➔ A classes derivadas ou subclasses podem ter acesso a métodos das superclasses usando a palavra-chave ***base***.
- ➔ Quando a superclasse contém construtores com parâmetros, eles devem ser chamados pelos construtores da subclasse.
- ➔ 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().*

Exemplo de Herança e Construtor

```
using System;
class Pai
{
    public Pai(string nome)
    {
        Console.WriteLine("Construtor Pai: {0}", nome);
    }
}
```

```
using System;
class Filho : Pai
{
    public Filho(string nome) : base(nome)
    {
        Console.WriteLine("Construtor Filho");
    }
}
```

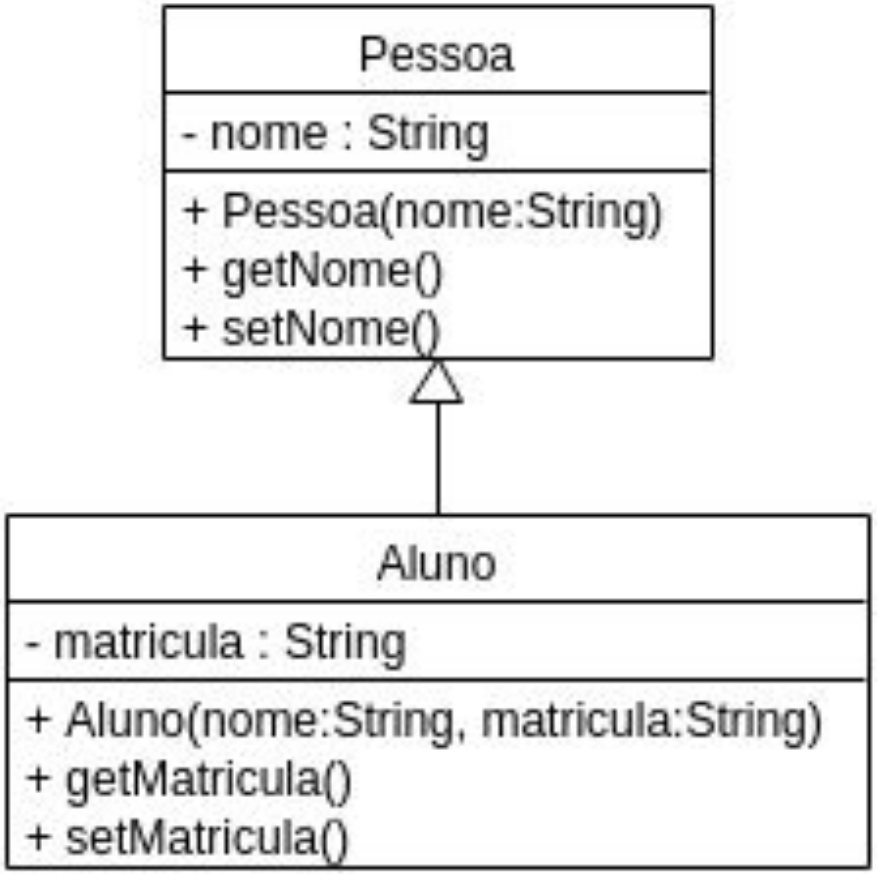
Exemplo de Herança e Construtor

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

Saída

```
Construtor Pai: Junior
Construtor Filho
```

Exemplo de Herança II



Reescrita (ou Sobrescrita) de Métodos

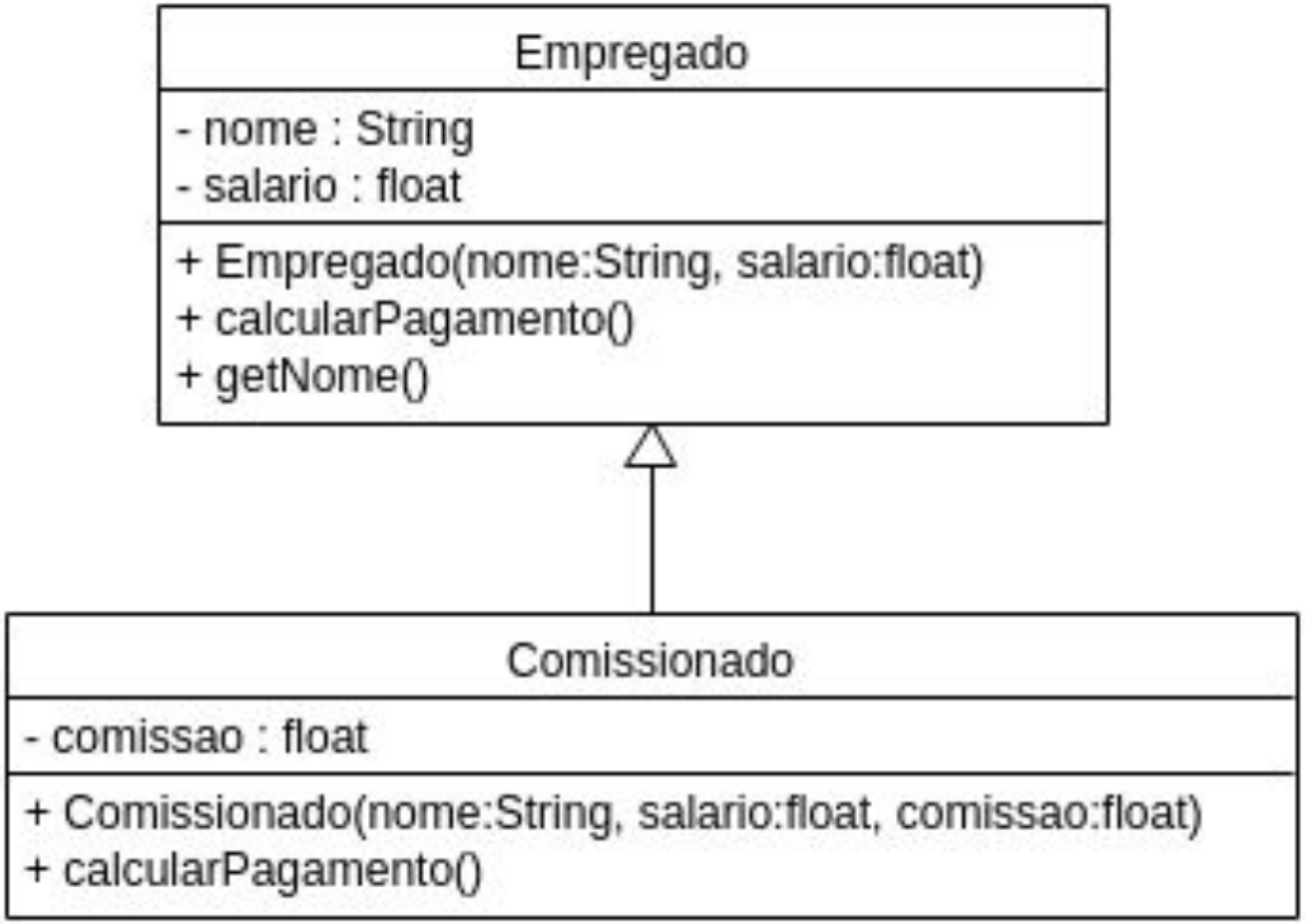
- É a definição de métodos com a mesma assinatura (nome e parâmetros) na classe base e em alguma classe derivada.
- Os métodos reescritos na classe subclasse têm prioridade sobre os métodos da superclasse.
  - Se o método chamado existir na subclasse ele é que será chamado, se não existir o método será procurado na superclasse.
- A razão de reescrever o método é que os métodos da subclasse geralmente executam tarefas adicionais que os mesmo métodos da superclasse.

Reescrita de Métodos em C#

➔ Por padrão em C#, as implementações dos métodos de uma superclasse não podem ser substituídas pelas subclasses.

- Para alterar esse padrão, devemos acrescentar o modificador ***virtual*** no método da superclasse, e o modificador ***override*** no método da subclasse que tem a mesma assinatura.

Exemplo de Herança III



Classes Abstratas

- ➔ Uma classe abstrata é uma classe que não pode ser instanciada diretamente.
- ➔ É uma classe base para outras classes. Portanto, ela deve ser derivada.
- ➔ Normalmente são utilizadas para organizar e simplificar uma hierarquia de generalização.
- ➔ Propriedades comuns a diversas classes podem ser organizadas e definidas em uma única classe da qual as primeiras herdam.

Classes Abstratas

- ➔ Uma classe abstrata pode conter métodos abstratos.
- ➔ Uma classe não abstrata derivada de uma classe abstrata deve incluir implementações reais de todos os métodos abstratos herdados.
- ➔ Em C#, uma classe abstrata é definida utilizando-se o modificador ***abstract*** antes da declaração da classe.
- ➔ Exemplo:

```
abstract class Funcionario {...}
```

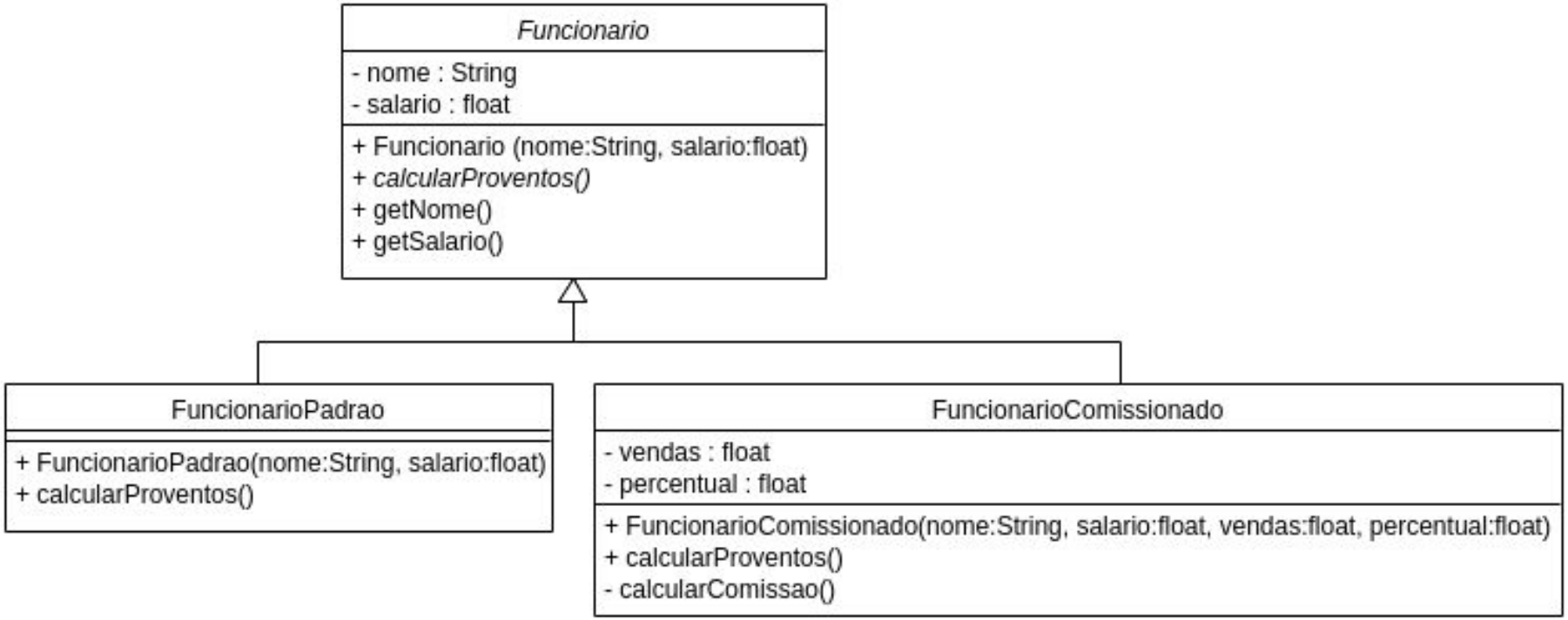
Métodos Abstratos

- ➔ Não fornece nenhuma implementação real, não há nenhum corpo de método.
- ➔ Um método abstrato é implicitamente um método virtual.
- ➔ Declarações de método abstrato são permitidas apenas em classes abstratas.
- ➔ A declaração de um método abstrato é feita utilizando-se a palavra ***abstract*** antes da definição do método.
  - ◆ A declaração do método abstrato termina com um ponto e vírgula e não há chaves ({ }) após a assinatura.
- ➔ A implementação é fornecida por um método ***override***, que representa um membro de uma classe não abstrata.

Exemplo - Classe Abstrata

- ➔ Cadastrar os funcionários de uma empresa e calcular os proventos devidos a cada um deles por um mês de trabalho.
- ➔ Proventos: representam quaisquer valores devidos aos funcionários a título de remuneração.
- ➔ Serão considerados apenas dois tipos de proventos:
  - ◆ Salário Fixo: devido a todos os funcionários.
  - ◆ Comissão: adicional devido pelas vendas realizadas pelos funcionários que são comissionados.

Exemplo - Classe Abstrata



Polimorfismo

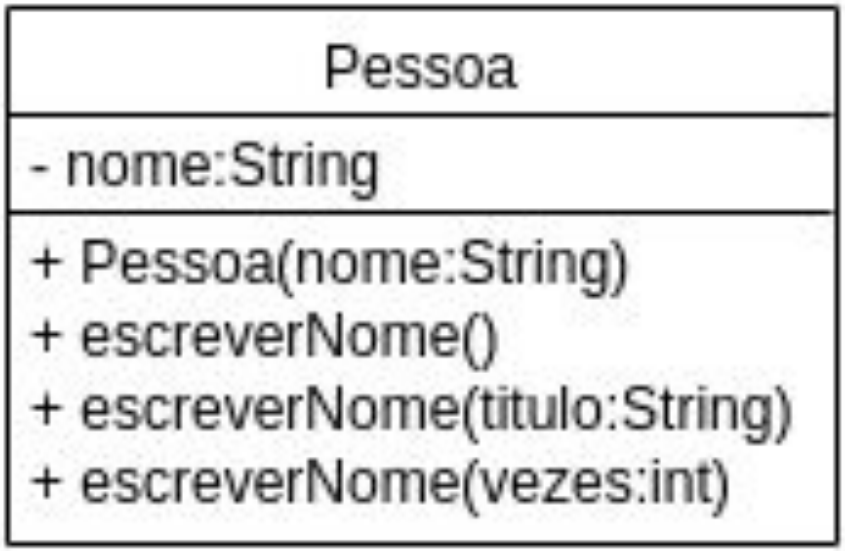
- ➔ O termo polimorfismo significa “muitas formas”.
- ➔ Polimorfismo significa que uma mesma operação pode se comportar de maneira diferente dependendo do estímulo recebido e da classe onde se encontra.
- ➔ Uma operação pode ser implementada de muitas formas.
- ➔ Tipos Básicos de Polimorfismo
  - ◆ Sobrecarga
  - ◆ Sobrescrita (Substituição)

Sobrecarga (*Overload*)

- Mecanismo que permite definir duas ou mais métodos como o mesmo nome, porém o número ou o tipo de parâmetros são diferentes.
  - Pode-se definir tais operações tanto na classe base como nas classes derivadas.
- Exemplo:

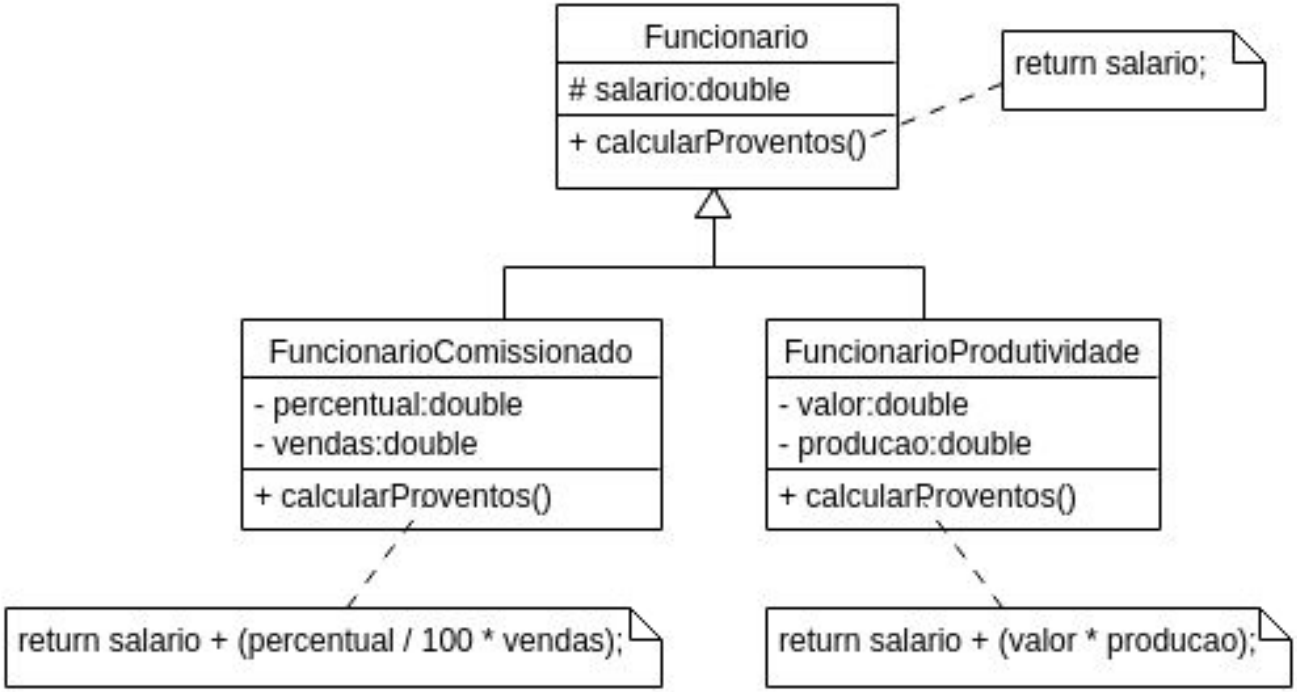
```
class HoraDia {
    void setHora (String hora);
    void setHora (int h, int m, int s);
}
...
HoraDia umRelogio;
umRelogio.setHora ("11:55:00");
umRelogio.setHora (17, 1, 0);
```

Exemplo Polimorfismo I



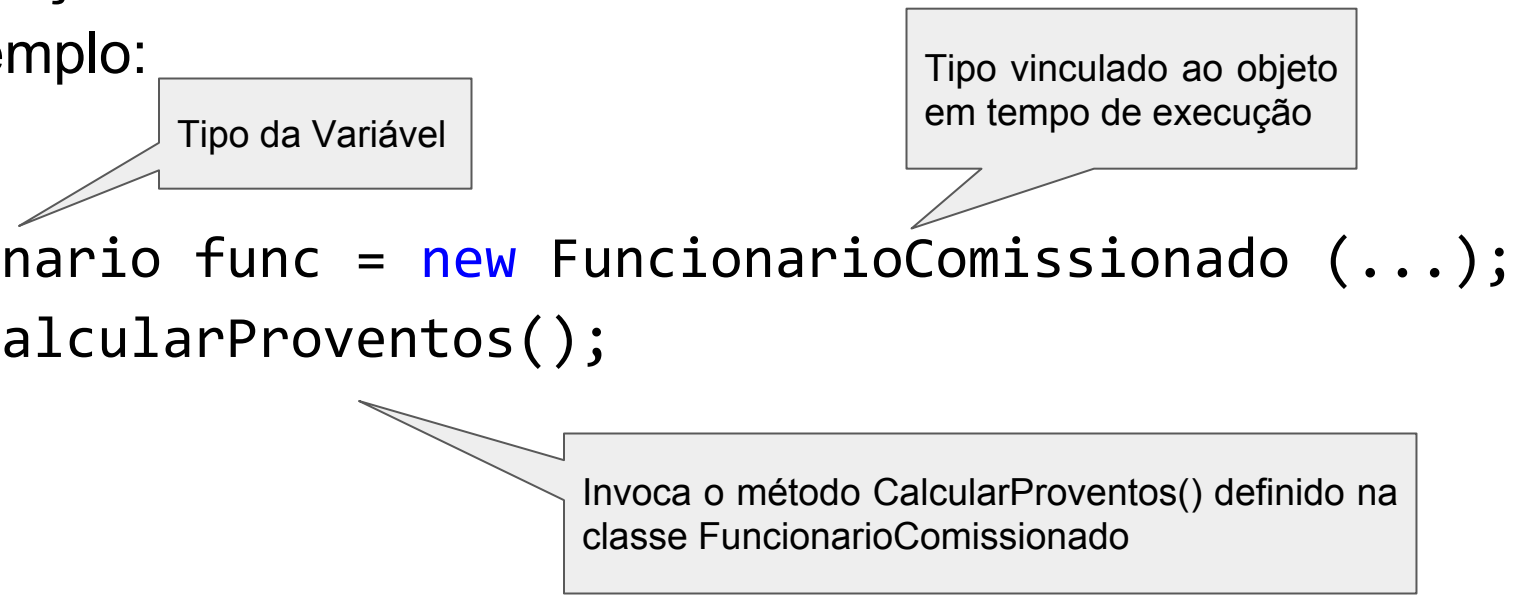
Sobrescrita (*Override*)

➔ Dizemos que uma operação em uma classe filha sobrescreve uma operação da classe pai se as duas operações têm o mesmo nome e os mesmos parâmetros.

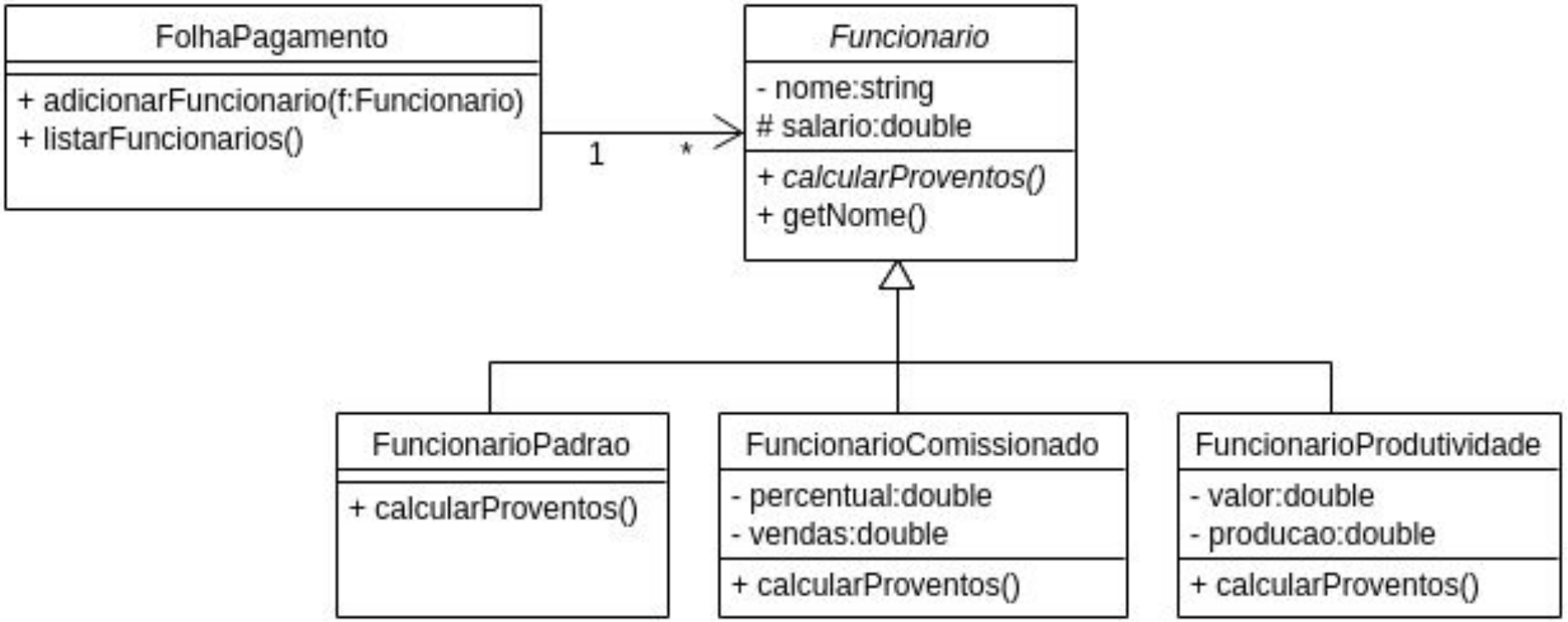


Vinculação Dinâmica (*Dynamic Binding*)

- ➔ O método a ser chamado somente é determinado em tempo de execução.
- ➔ O tipo da variável determina quais as operações que podem ser utilizadas, mas o tipo vinculado ao objeto determina que definição de método será utilizada.
- ➔ Exemplo:



Exemplo Polimorfismo II



Interface

- ➔ Uma interface em C# contém apenas as assinaturas de métodos, mas não fornece implementação para estes métodos (métodos abstratos).
- ➔ Classes podem implementar uma ou mais interfaces.
- ➔ Cada método da interface deve ser declarado em todas as classes que explicitamente implementam uma interface.
- ➔ Uma vez que uma classes implementa uma interface, todos os objetos daquela classe tem uma relação de *é-um* com o tipo da interface, e todos os objetos da classe garantem o fornecimento das funcionalidades descritas pela interface.

Interface

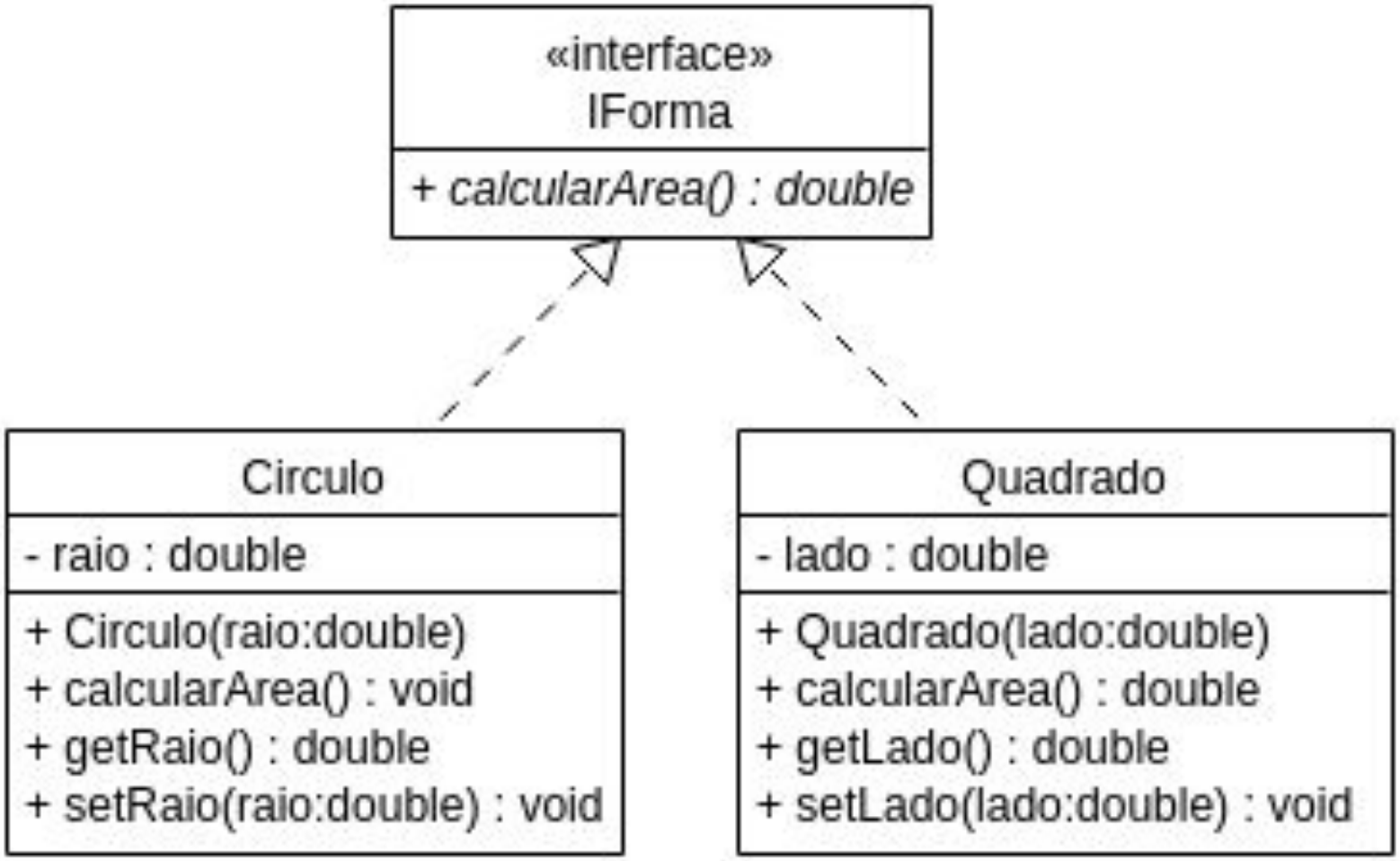
- ➔ Interfaces são particularmente úteis para a atribuição de funcionalidades comuns a classes possivelmente não relacionadas.
- ➔ Permite objetos de classes não relacionada serem processados polimorficamente.
- ➔ Uma interface é similar a uma classe, mas com algumas importantes diferenças:
  - ◆ Todas as operações são abstratas. Isto é, elas têm nome, parâmetros e um tipo de retorno, mas não têm nenhuma implementação.
  - ◆ Todas as operações de uma interface são automaticamente públicas.
  - ◆ Uma interface não tem atributos.
  - ◆ Uma interface não pode ser instanciada diretamente.

Interface em C#

- Em C# a declaração de uma interface é feita utilizando a palavra reservada *interface* seguida do nome dela.
  - Por estilo de codificação, o nome de uma interface começa com I.
- Para indicar que uma classe implementa uma interface, coloca-se dois ponto (:) e o nome da interface logo após a declaração da classe.
- Para implementar um membro de interface, o membro correspondente da classe de implementação deve ser público, não estático e ter o mesmo nome e assinatura do membro de interface.

```
interface IForma {...}
class Circulo : IForma {...}
```

Exemplo Interface I

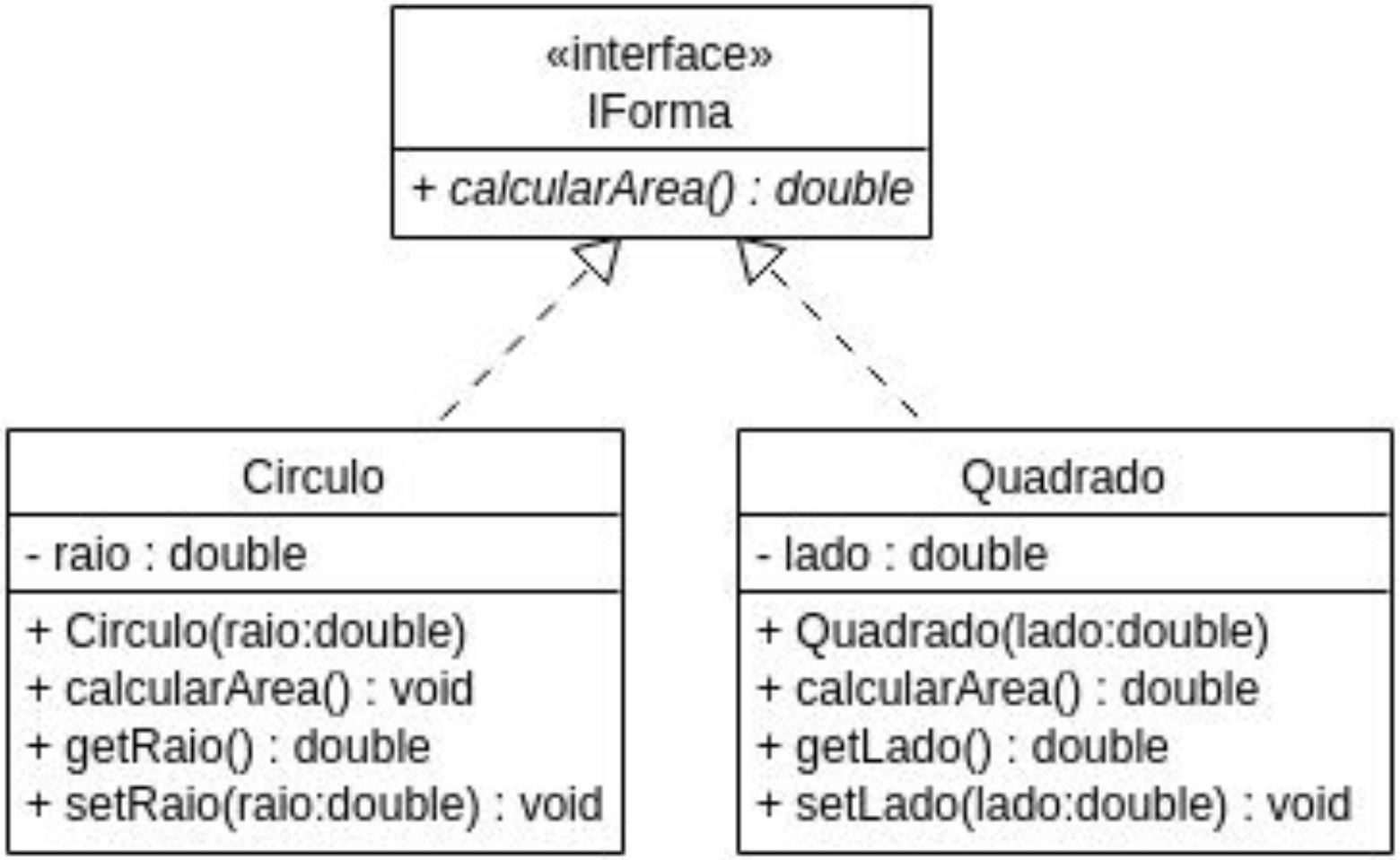


Interface em C#

➔ Uma classe que implementa uma interface pode implementar métodos dessa interface explicitamente.

- Um método implementado explicitamente não pode ser acessado por meio de uma instância da classe, mas apenas por meio de uma instância da interface.
- Isso é realizado ao nomear o membro da classe com o nome da interface e um ponto final.

Exemplo Interface II - Usando Interface Explícita



Exceções

- Exceção é uma indicação de um problema que ocorre durante a execução de um programa.
  - É um evento, normalmente associado a um erro, que ao ser disparado interrompe o fluxo normal de execução da aplicação.
  - Quando uma exceção ocorre, a seqüência normal do fluxo é finalizada e a rotina de tratamento de exceção é executada.
  - Exemplo: divisões por zero, acesso a posições inválidas em listas ou vetores, falta de memória, disco cheio, …
  - São situações que não ocorrem com frequência, mas aplicações devem estar preparadas para tratar essas exceções sem abortar o programa.
- O tratamento de exceções é um recurso para resolver exceções que poderiam ocorrer para que o programa continue ou termine elegantemente.
  - Permite que os programadores criem programas mais robustos e tolerantes a falhas.

Exceções

- ➔ O recurso de tratamento de exceção da linguagem C# ajuda a lidar com quaisquer situações excepcionais ou inesperadas que ocorram quando um programa for executado.
- ➔ O tratamento de exceção usa as palavras-chave **try**, **catch** e **finally** para executar ações que podem não ser bem-sucedidas, lidar com falhas quando decidir se é razoável fazer isso e limpar recursos posteriormente
- ➔ As exceções são lançadas usando a palavra-chave **throw**.
- ➔ Pode-se decidir quais exceções se quer tratar – todas as exceções, todas as exceções de um certo tipo, todas as exceções de um grupo de tipos relacionados.

Instruções

- **try**: bloco onde pode ocorrer um erro.
- **catch**: bloco onde será tratado o erro.
- **finally**: bloco que será sempre executado (opcional - útil quando se quer liberar um recurso, por exemplo).
- **throw**: instrução para forçar a ocorrência de uma exceção.

Tratamento de Exceções em C#

```
try {
    // Instruções a serem executadas
}
catch (TipoExceção erro) {
    // Tratamento de exceções de um tipo específico
}
catch (Exception erro) {
    // Tratamento genérico de exceções
}
finally {
    // Instruções sempre executadas, ocorram ou não erros
}
```

Exemplo - Divisão por Zero sem Exceção

```
using System;
namespace ExemploExcecaoI
{
   class Program
   {
       static void Main(string[] args)
       {
           int n = 10;
           int d = 0;
           int divisao = n / d;
           Console.WriteLine("Resultado: {0}", divisao);
       }
   }
}
```

Saída

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at ExemploExcecaoI.Program.Main(String[] args) in Program.cs:line 11
```

Exemplo - Tratando a Exceção

```
using System;
namespace ExemploExcecaoI
{
   class Program
   {
       static void Main(string[] args)
       {
           int n = 10;
           int d = 0;
           try
           {
               int divisao = n / d;
               Console.WriteLine("Resultado: {0}", divisao);
           }
           catch(ArithmeticException ae)
           {
               Console.WriteLine("Erro: Divisao por zero!");
           }
       }
   }
}
```

Saída

```
Erro: Divisao por zero!
```

Bloco *Try*

➔ O bloco *try* é onde deve ser implementado o fluxo principal do programa.

- O bloco try é obrigatório;
- O C# tentará executar todas as instruções no bloco *try*;
- Se nenhuma instrução gerar uma exceção, todas serão executadas;
- Se ocorrer algum erro, a execução é pulada para os blocos *catch* ou *finally*. As instruções restantes no *try* são ignoradas;
- Se nenhum *catch* capturar a exceção, ela é capturada pelo CLR e a execução do aplicativo pode ser encerrada, dependendo do tipo de erro.

Bloco *Catch*

➔ O bloco *catch* é onde devem ser implementados os tratamentos de erro.

- O bloco *catch* é opcional;
- Um bloco *try* pode possuir vários blocos *catch*;
- Os manipuladores devem ser escritos do mais específico para o mais genérico;
- A classe mais genérica de erro é a *Exception*;
- A exceção é tratada no bloco *catch*, ou seja, a priori apenas um catch é executado.

Bloco *Finally*

➔ No bloco *finally* são incluídas as instruções que devem necessariamente ser executadas, ocorram ou não erros.

- O bloco *finally* é opcional;
- Um bloco try possui apenas um bloco *finally*;
- Ele deve ser o último bloco da estrutura;
- Blocos *finally* são utilizados para liberação de recursos como manipuladores de arquivos e conexões com bancos de dados;
- Numa execução sem erros são executados os blocos *try* e *finally*;
- Numa execução com erros são executados os blocos *try* (parcialmente), *catch* (de acordo com o tipo do erro) e *finally* (sempre).

Exemplo Exceção II

```
using System;
namespace ExemploExcecaoII
{
   class Program
   {
       static void Main(string[] args)
       {
           try
           {
               Console.Write("Numerador: ");
               int n = int.Parse(Console.ReadLine());
               Console.Write("Denominador: ");
               int d = int.Parse(Console.ReadLine());
               Console.WriteLine("Divisao: {0}", n / d);
           }
           catch(DivideByZeroException e)
           {
               Console.WriteLine("Erro: {0}", e.Message);
           }
           catch(FormatException e)
           {
               Console.WriteLine("Erro: {0}", e.Message);
           }
           finally
           {
               Console.WriteLine("Fim do programa.");
           }
       }
   }
}
```

Exemplo Exceção II - Saída

Execução 01

```
Numerador: A
Erro: Input string was not in a correct format.
Fim do programa.
```

Execução 02

```
Numerador: 10
Denominador: 0
Erro: Attempted to divide by zero.
Fim do programa.
```

Execução 03

```
Numerador: 10
Denominador: 5
Divisao: 2
Fim do programa.
```

Criando Exceções em C#

- ➔ Em C#, é possível criar exceções personalizadas.
- ➔ Roteiro para criar uma exceção:
  - ◆ Criar uma classe (chamar de xxxException), onde xxx é o tipo da exceção (ex.: MinhaException) e derivá-la de *Exception* (System.Exception);
  - ◆ Criar construtores que chamam os construtores de *Exception*.
  - ◆ Utilizar a exceção no código.
- ➔ Para lançar uma exceção utiliza-se a palavra reservada **throw** seguida de uma instância da exceção que se quer lançar.

Exemplo Exceção III

```
using System;
namespace ExemploExcecaoIII
{
   class ValorInvalidoException : Exception
   {
       public ValorInvalidoException() { }

       public ValorInvalidoException(string message) : base(message) { }
   }
}
```

```
using System;
namespace ExemploExcecaoIII
{
   class Program
   {
       static void Main(string[] args)
       {
          int n = 10;
          int d = 0;
          try
          {
              Console.WriteLine("Resultado: {0}", divisao(n, d));
          }
          catch(ValorInvalidoException e)
          {
              Console.WriteLine("Erro: {0}", e.Message);
          }
          finally
          {
              Console.WriteLine("Fim do Programa.");
          }
       }

       static int divisao(int n, int d)
       {
           if (d == 0)
           {
               // lança explicitamente uma exceção
               throw new ValorInvalidoException("O denominador deve ser diferente de zero.");
           }
           return n / d;
       }
   }
}
```