Programação Orientada a Objetos

Módulo 2

Introdução à Programação Orientada a Objetos

Curso de Gestão da Informação

Prof. Dr. Daniel A. Furtado - FACOM/UFU

Objetivos do Módulo

- Apresentar um breve histórico da Programação Orientada a Objetos.
- Introduzir a Programação Orientada a Objetos.
- Introduzir o conceito de classe e objeto.
- Introduzir o conceito de atributos, propriedades, métodos e construtores.
- Introduzir aspectos de visibilidade e modificadores de acesso.
- Introduzir o conceito de sobrecarga de método e construtor.
- Introduzir o conceito de membros de instância e membros de classe.
- Introduzir o conceito de encapsulamento.
- Introduzir a representação de objetos em memória.

Histórico da POO - Simula

- Simula é considerada a primeira linguagem orientada a objetos.
- Surgiu em 1967 na Noruega para simular eventos do mundo real.
- Introduziu conceitos básicos como Classes e Objetos.
- Introduziu a ideia de Garbage Collection.

Histórico da POO - Smalltalk

- No início dos anos 1970 foi desenvolvida a Smalltalk no Xerox PARK.
- Traz a ideia de orientação à objeto pura – tudo é objeto.
- Popularizou a ideia de "mensagens" entre objetos (chamada de método).
- Introduziu os conceitos de Encapsulamento, Herança e Polimorfismo.
- Implementa o conceito de máquina virtual (antecessor da JVM).

Histórico da POO - C++

- No início dos anos 1980 surge o C++.
- Adiciona classes e objetos à linguagem C.
- Torna a POO viável no desenvolvimento de aplicações de alto desempenho e sistemas operacionais.
- Primeira linguagem orientada a objetos amplamente utilizada pela indústria.

Histórico da POO - Java

- Em 1995 a Sun Microsystems lança a linguagem Java
- Projetada para ser robusta e portável
  - *"Write Once, Run Anywhere*".
  - A Java Virtual Machine (JVM) permitiu que o mesmo código pudesse ser executado em diferentes sistemas operacionais.
- Popularização da POO em aplicações empresariais.

Histórico da POO – C#

- Lançada pela Microsoft em 2002 como parte do *framework* .NET.
- Projetada para ser uma evolução das linguagens Java e C++.
- Puramente orientada a objetos, mas com um design mais moderno.
- Código aberto e multiplataforma.
- Permite o desenvolvimento de aplicativos de diversos tipos para Windows, macOS, Linux, iOS e Android.

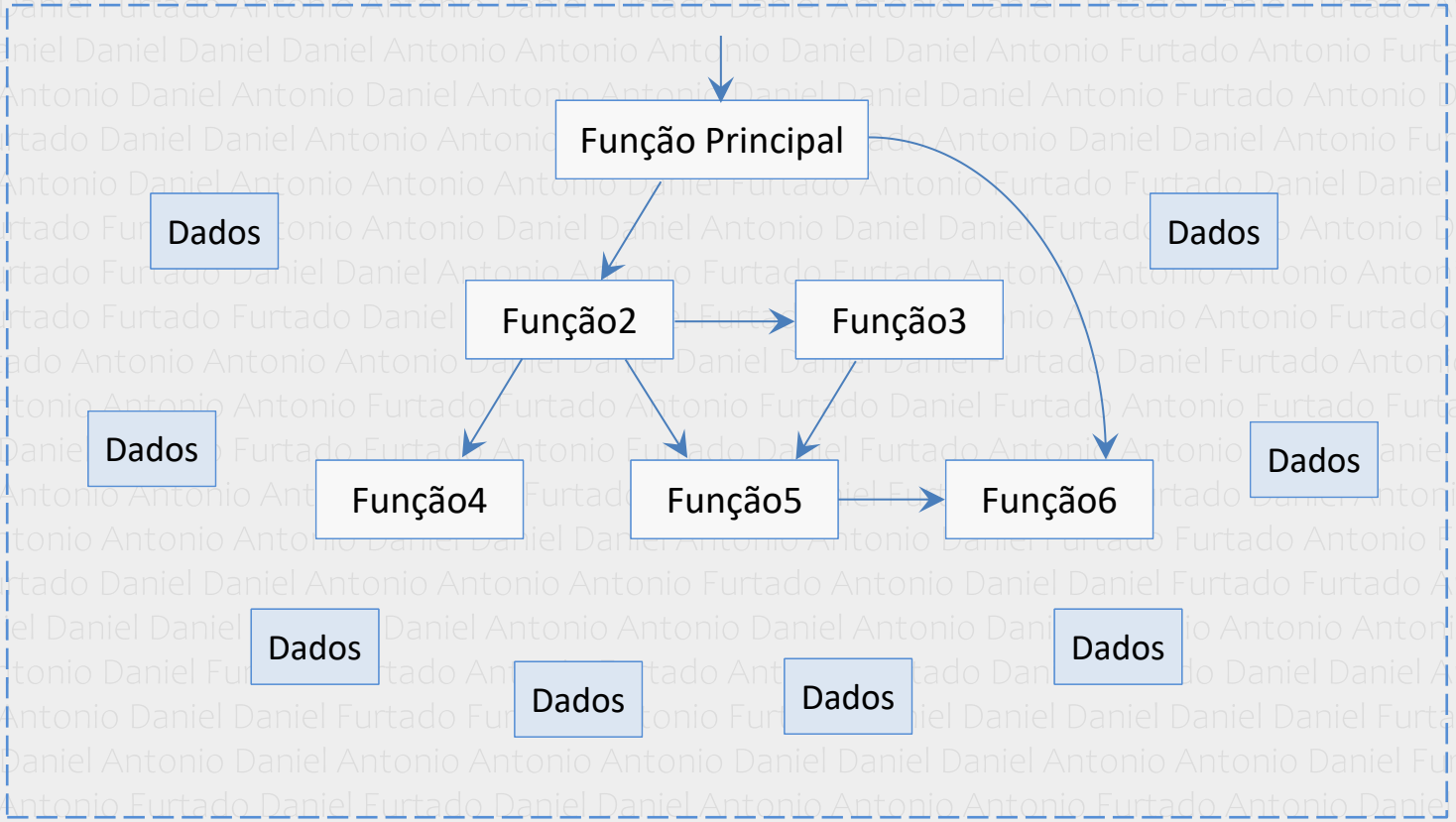
Histórico da POO - Atualidade

- Atualmente, a POO constitui o paradigma de programação dominante no desenvolvimento de software.
- Várias linguagens passaram a ter algum suporte à POO, mesmo sem ser puramente orientada a objetos:
  - JavaScript, Python, Swift, PHP, Kotlin etc.
  - Integração da POO com outros paradigmas como o funcional, procedural e outros.

Características da Programação Procedural

- Código baseado em funções e rotinas.
- Dados são separados dos comportamentos (funções).
- Reutilização de código limitada.
- Simples para programas pequenos, porém difícil de manter e desenvolver em sistemas grandes e complexos.

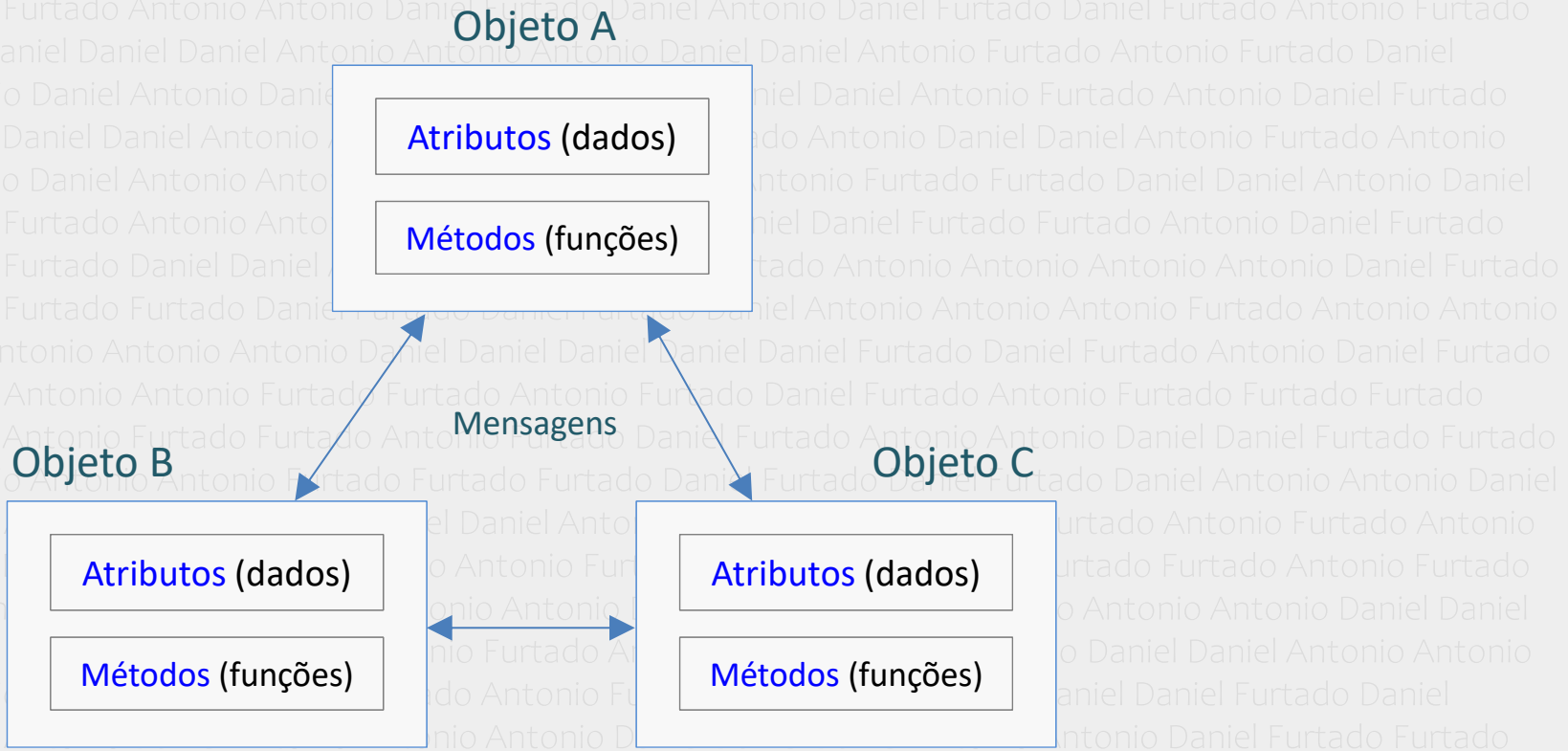
Programação Procedural



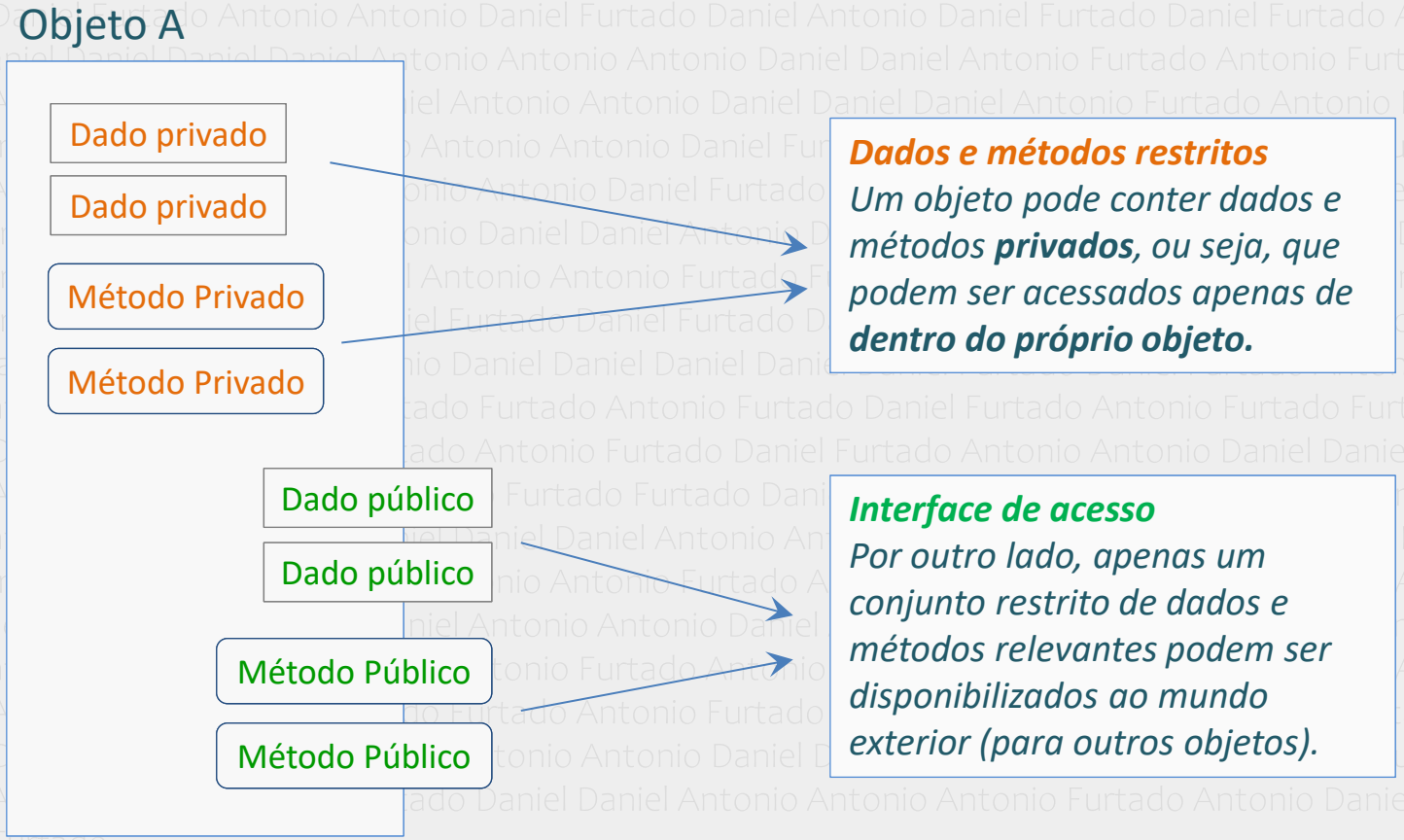
Programação Orientada a Objetos (POO)

- Baseada na ideia de se criar objetos para representar entidades ou conceitos do mundo real como cliente, produto, conta bancária etc.
- Objetos incluem seus próprios dados (denominados atributos) e suas próprias funções que manipulam esses dados (denominadas métodos).
  - Dados e comportamentos combinados.
- Objetos podem se comunicar uns com os outros, mas detalhes de implementação não precisam ser expostos entre eles.
  - Encapsulamento: abstração e proteção dos dados.
- Um programa em POO é basicamente uma coleção de objetos comunicando uns com os outros.
- Maior facilidade de manutenção e reutilização de código.

Programação Orientada a Objetos (POO)



Programação Orientada a Objetos (POO)



Classes

- Uma classe é basicamente uma definição da estrutura do objeto, incluindo seus atributos e métodos:
  - Os atributos representam dados (características) que os objetos terão.
  - Os métodos representam comportamentos (ações) que os objetos deverão realizar.
- A classe tem papel similar a um molde ou fôrma, pois ela é utilizada para criação dos objetos propriamente ditos.
  - O molde define o formato e as características que as peças terão.
  - Porém o molde em si não é a peça. É usado para criá-la.
- Uma classe corresponde à definição de um novo tipo de dados.

Classes

- Suponha que seja necessário desenvolver um jogo em 2D que apresente formas geométricas na tela como círculos, triângulos e quadrados.
- Nesse contexto, poderíamos pensar na criação de uma classe Circulo contendo atributos e métodos relevantes para exibição na tela, como:
  - Atributos:
    - raio,
    - cor,
    - posicaoX,
    - posicaoY
  - Métodos:
    - MoverParaBaixo(),
    - MoverParaDireita(),
    - Ampliar(), Reduzir(),
    - Exibir()

Classes

- Suponha que seja necessário desenvolver um aplicativo para edição e apresentação de fotos.
- Nesse contexto, poderíamos criar uma classe Foto contendo:
  - Atributos:
    - nomeArquivo,
    - larguraEmPixels,
    - alturaEmPixels,
    - formato,
    - matrizDePixels
  - Métodos:
    - Exibir(),
    - Redimensionar(int novaLargura, int novaAltura),
    - Clarear(),
    - Escurecer()
    - Converter()

Objetos da Classe

- Definir a classe é apenas o primeiro passo.
- A partir da classe podemos criar inúmeros objetos da classe.
  - Um objeto é uma instância da classe.
- No exemplo do jogo 2D, poderíamos ter vários círculos diferentes se movimentando na tela, onde cada círculo é um objeto da classe.
- Cada objeto ocupa espaço de armazenamento próprio, e possui seus próprios valores para os atributos definidos na classe.
- Os métodos frequentemente alteram os atributos do objeto, causando uma mudança de seu estado.
- Portanto, cada objeto é único e independente, com seu próprio estado (definido pelos valores correntes dos atributos).

Objetos da Classe

- circulo1, com os atributos a seguir, representa um objeto da classe Circulo
  - raio: 15,
  - cor: "laranja",
  - posicaoX: 100,
  - posicaoY: 80
- circulo2, com os atributos a seguir, representa outro objeto da classe Circulo
  - raio: 48,
  - cor: "azul",
  - posicaoX: 20,
  - posicaoY: 30

Objetos da Classe

- foto1, com os atributos a seguir, representa um objeto da classe Foto:
  - nomeArquivo: praia.jpg,
  - larguraEmPixels: 1920,
  - alturaEmPixels: 1080,
  - formato: jpeg
- foto2 é representa outro objeto da classe Foto:
  - nomeArquivo: paisagem.jpg,
  - larguraEmPixels: 3840,
  - alturaEmPixels: 2160,
  - formato: jpeg

Definição de Classes em C#

- Classes em C# são definidas com a palavra reservada class seguida pelo nome da classe.
- O corpo da classe deve ser colocado entre chaves { }.
- No corpo são definidos os atributos e os métodos.
- Os atributos são comumente definidos no início da classe, com sintaxe similar à sintaxe de definição de variáveis.
- Os métodos são definidos de forma similar às funções:

```
<modificadorDeAcesso> tipoDeRetorno NomeDoMetodo(parâmetros)
```

```
class NomeDaClasse
{
    // Atributos


    // Métodos
}
```

Definição de Classes em C# - Exemplo Introdutório

```
class Circulo
{
    // Atributos
    int raio;
    int posX;
    int posY;

    // Métodos
    public void MoverParaDireita() { posX += 1; }
    public void MoverParaBaixo() { posY += 1; }
    public void Ampliar() { raio += 1; }
    public void Exibir()
    {
        Console.WriteLine($"Raio: {raio}, Posição: {posX},{posY}");
    }
}
```

Todos os objetos do tipo Circulo criados a partir dessa classe introdutória teriam atributos com valores iniciais iguais a zero.

OBS: frequentemente o termo campo (*field*) ou variável membro é utilizado para fazer referência à variável associada ao atributo.

Criação de Objetos da Classe

```
// Cria dois objetos da classe Circulo, c1 e c2
// raio, posX e posY serão inicializados com zero
var c1 = new Circulo();
var c2 = new Circulo();

// Chamada dos métodos no objeto c1
c1.Ampliar(); // Incrementa o raio de c1
c1.Ampliar(); // Incrementa novamente o raio de c1
c1.MoverParaDireita(); // Incrementa a posição x de c1

// Chamada dos métodos no objeto c2
c2.Ampliar(); // Incrementa o raio de c2
c2.MoverParaBaixo(); // Incrementa a posição y de c2

c1.Exibir(); // Mostra "Raio: 2, Posição: 1,0"
c2.Exibir(); // Mostra "Raio: 1, Posição: 0,1"
```

OBS: No lugar da palavra var poderia ser utilizado o nome da classe de forma similar à declaração de variáveis de um determinado tipo, ou seja: Circulo c1 = new Circulo();

Métodos de Instância e Criação de Objetos

- Os métodos definidos na classe Circulo são denominados métodos de instância, pois precisam ser chamados utilizando um objeto da classe.
- Tais métodos executam ações no objeto em particular onde ele é chamado.
- Sintaxe para chamar um método de instância:

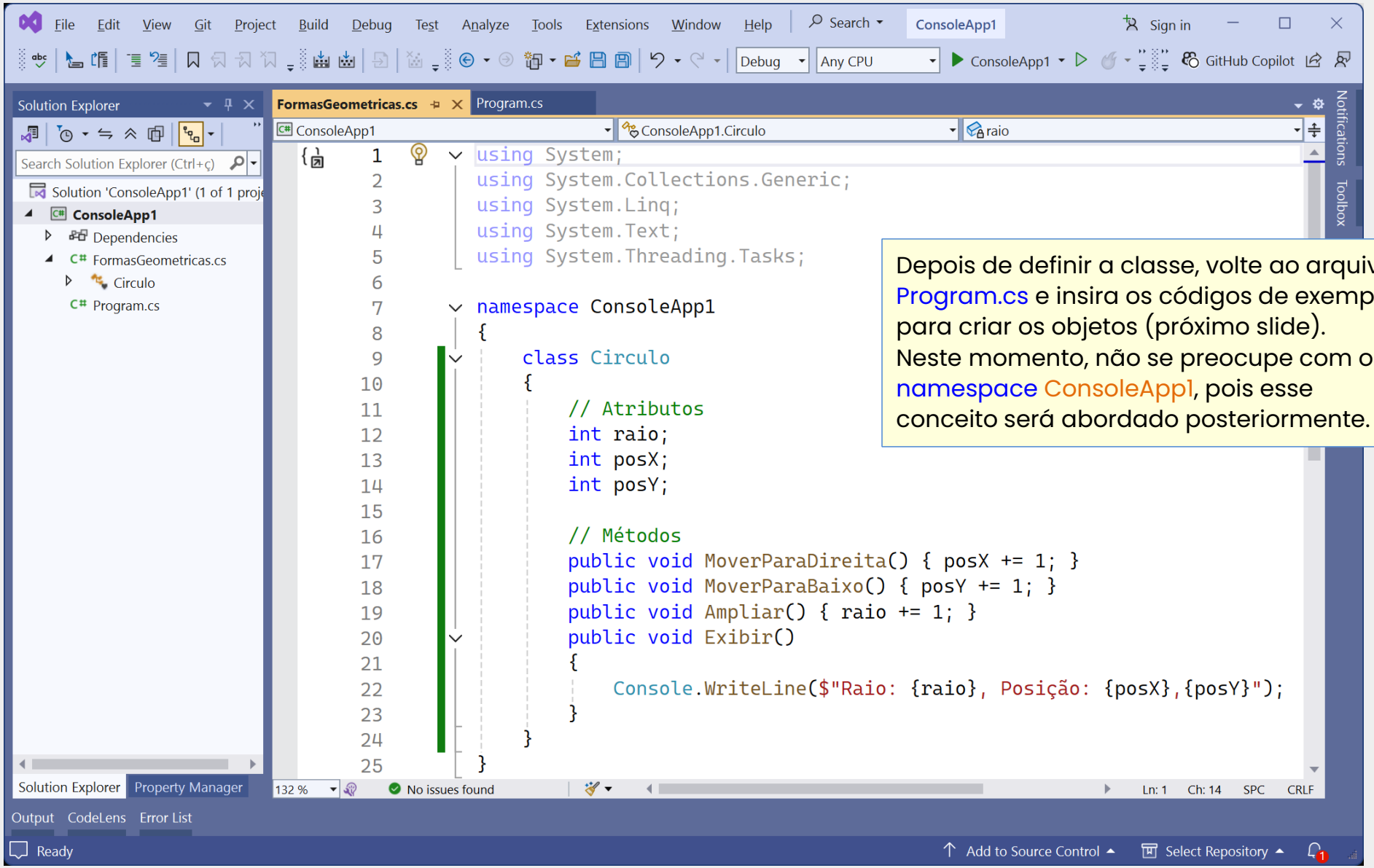
```
nomeDoObjeto.NomeDoMetodo(argumentos)
```

- Para criar novos objetos da classe definida previamente pode-se utilizar a palavra reservada new, conforme apresentado a seguir.

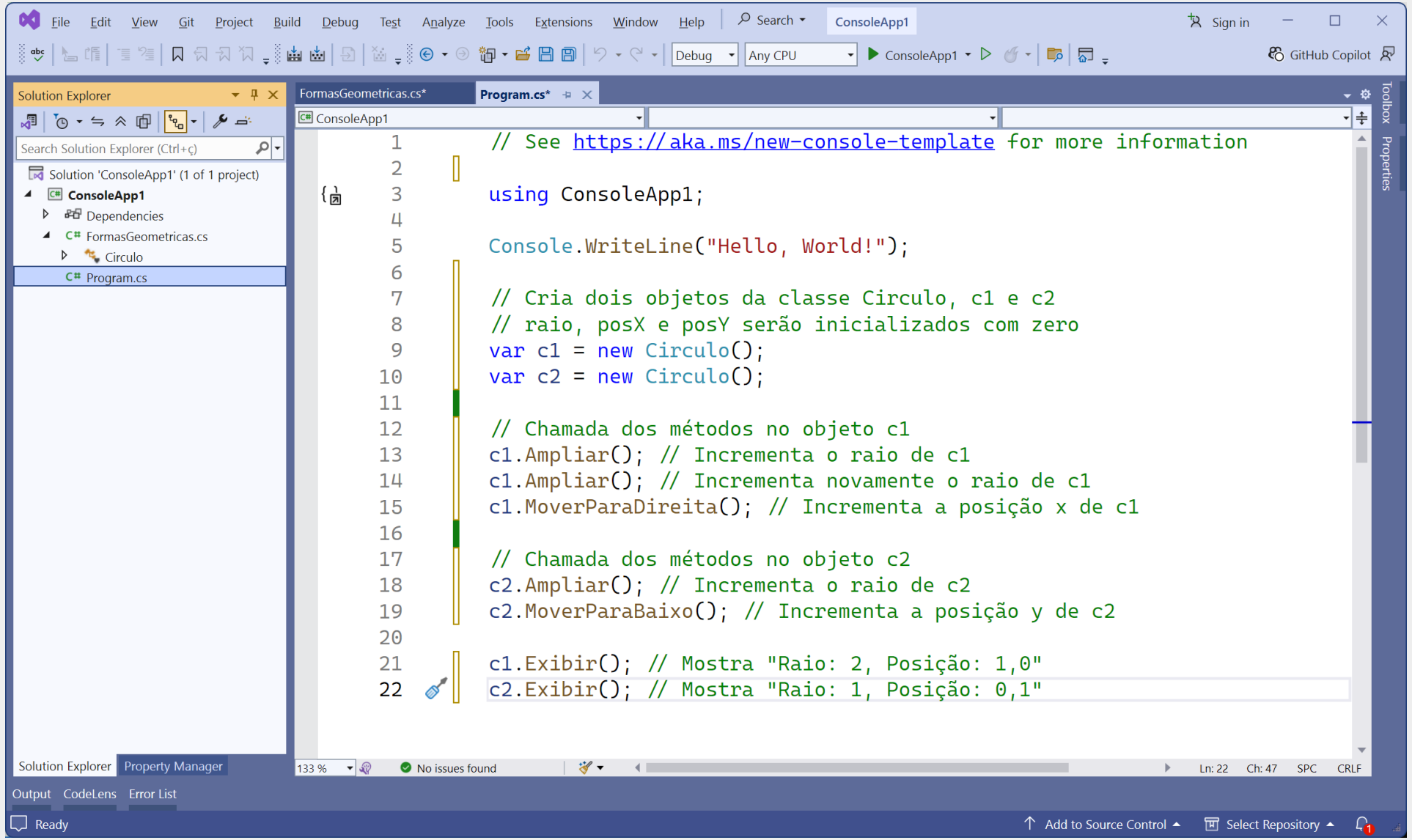
Criação de Classes no Visual Studio Community

- Em um projeto no Visual Studio Community, recomenda-se definir novas classes fora do arquivo Program.cs.
- Para adicionar um novo arquivo de código C# ao projeto, clique com o botão direito sobre o nome do projeto dentro da janela de ferramentas Solution Explorer, escolha Add → New Item e informe um nome para o novo arquivo (ex. FormasGeometricas.cs).
- Em seguida, altere o conteúdo dos arquivos conforme slides a seguir.

Criação de Classes no Visual Studio Community



Criação de Classes no Visual Studio Community



Método Construtor

- Com o exemplo anterior, todo novo objeto da classe Circulo teria inicialmente o valor padrão 0 para os seus atributos.
- Esse aspecto pode ser aprimorado para permitir que novos objetos sejam criados com valores específicos para os atributos.
- Para isso, é preciso definir um método especial na classe denominado método construtor (ou simplesmente construtor).
- O construtor será automaticamente chamado ao utilizar a palavra reservada new para criar um novo objeto.
- O construtor não possui tipo de retorno e deve ter o mesmo nome da classe.

Além da mera inicialização dos atributos, o método construtor geralmente tem uma responsabilidade maior e pode incluir operações adicionais para garantir que o objeto seja criado em um estado consistente e esteja pronto para uso.

Definição de Classes com Construtor

```
class Circulo
{
    // Atributos
    int raio;
    int posX;
    int posY;

    // Método construtor com três parâmetros
    public Circulo(int r, int x, int y)
    {
        raio = r; // inicializa o atributo raio
        posX = x; // inicializa o atributo posX
        posY = y; // inicializa o atributo posY
    }

    public void MoverParaDireita() { posX += 1; }
    public void MoverParaBaixo() { posY += 1; }
    public void Ampliar() { raio += 1; }
};
```

OBS: esta é a sintaxe clássica de construtores em C#. Porém, a partir do C# 12, há também a sintaxe concisa de construtor primário, que será apresentada posteriormente.

Utilizando o Construtor na Criação de Objetos

```
// Cria dois objetos da classe Circulo, c1 e c2
// c1 terá um raio inicial de 5 e posição inicial 10,15
// c2 terá um raio inicial de 8 e posição inicial 20,25
// O construtor será chamado automaticamente
var c1 = new Circulo(5, 10, 15);
var c2 = new Circulo(8, 20, 25);

// Chamada dos métodos no objeto c1
c1.Ampliar(); // Incrementa o raio de c1 para 6
c1.MoverParaDireita(); // Incrementa a posição x de c1 para 11

// Chamada dos métodos no objeto c2
c2.Ampliar(); // Incrementa o raio de c2 para 9
c2.MoverParaBaixo(); // Incrementa a posição y de c2 para 26
```

Sobrecarga de Construtor

- Uma classe pode ter vários construtores, desde que suas assinaturas (quantidade e/ou tipo dos parâmetros) sejam diferentes.
- Isso é chamado de sobrecarga de construtor.
- Pode dar maior flexibilidade e praticidade na inicialização do objetos (conforme necessidades específicas das circunstâncias de uso).

Sobrecarga de Construtor

```
class Circulo
{
    // Atributos
    int raio;
    int posX;
    int posY;

    // Método construtor com um único parâmetro
    public Circulo(int r)
    {
        // Inicializa o raio com o parâmetro,
        // mas as posições são inicializadas com 0.
        raio = r;
        posX = 0;
        posY = 0;
    }

    // Método construtor com três parâmetros
    public Circulo(int r, int x, int y)
    {
        raio = r; // inicializa o atributo raio com o parâmetro r
        posX = x; // inicializa o atributo posX
        posY = y; // inicializa o atributo posY
    }
}
```

Sobrecarga de Construtor

```
// Cria o objeto c1 utilizando o primeiro construtor
var c1 = new Circulo(5);

// Cria o objeto c2 utilizando o segundo construtor
var c2 = new Circulo(8, 20, 25);

c1.Exibir(); // Mostra "Raio: 5, Posição: 0,0"
c2.Exibir(); // Mostra "Raio: 8, Posição: 20,25"
```

Construtor Padrão

- Quando nenhum construtor é definido de forma explícita, como no primeiro exemplo, um construtor padrão é criado automaticamente pelo compilador.
- O construtor padrão, nesse caso, não possui parâmetros e fará a inicialização dos atributos utilizando valores padrão (como o 0).

Sobrecarga de Método

- Além dos construtores, também é possível que vários métodos da classe compartilhem o mesmo nome, desde que suas assinaturas sejam diferentes.
  - Quantidade e/ou tipo dos parâmetros diferentes
- Isso é denominado sobrecarga de método.

```
class Circulo
{
    int raio; int posX; int posY;

    // Construtor
    public Circulo(int r, int x, int y) {
        raio = r;
        posX = x;
        posY = y;
    }

    // Se apenas uma coordenada é informada,
    // move o círculo na horizontal
    public void Mover(int x) {
        posX += x;
    }

    // Se as duas coordenadas são informadas
    // move o círculo nas duas direções
    public void Mover(int x, int y) {
        posX += x;
        posY += y;
    }
};
```

Exemplo de sobrecarga do método Mover

Nomes de Classes, Atributos e Métodos

- Por convenção, recomenda-se que os nomes de classes e métodos em C# sigam o padrão PascalCase.
  - A primeira letra de cada palavra aparece em maiúsculas.
  - Exemplos: ClienteRemoto, Usuario, GerarPedido(), AdicionarItem()
- Os nomes de atributos, variáveis e parâmetros devem seguir o padrão camelCase, que é similar ao PascalCase, porém com a primeira letra da primeira palavra em minúscula.
  - Exemplos: raio, nome, saldoCliente, valorDoPedido

Exercício

- Defina uma classe para representar veículos. Deve haver atributos para armazenar a marca, o modelo, o ano e a velocidade do veículo.
  - O construtor deve permitir a criação de novos objetos informando a marca, o modelo e o ano.
  - A velocidade inicial de todo veículo deve ser 0.
  - Deve haver um método para acelerar o veiculo e outro para pará-lo instantaneamente.
- No "programa principal", crie três instâncias da classe e mostre exemplos de chamadas dos métodos.

Palavra Reservada this

- No exemplo a seguir, seria possível criar parâmetros no construtor com os mesmos nomes dos atributos? Haveria algum conflito?

```
class Circulo
{
    // Atributos
    int raio;
    int posX;
    int posY;

    // Método construtor
    public Circulo(int r, int x, int y)
    {
        raio = r; // inicializa o atributo raio com o parâmetro r
        posX = x; // inicializa o atributo posX
        posY = y; // inicializa o atributo posY
    }

    // Métodos de instância
    public void MoverParaBaixo() { posX += 1; }
    public void MoverParaDireita() { posY += 1; }
    public void Ampliar() { raio += 1; }
}
```

Palavra Reservada this

- Nesse tipo de situação, é necessário utilizar a palavra reservada this para fazer referência ao raio que é atributo do objeto, diferenciando do parâmetro raio.
- this é sempre utilizada para fazer referência ao objeto em questão.

```
class Circulo
{
    // Atributos
    int raio;
    int posX;
    int posY;

    // Método construtor com três parâmetros
    public Circulo(int raio, int posX, int posY)
    {
        this.raio = raio; // inicializa o atributo raio
        this.posX = posX; // inicializa o atributo posX
        this.posY = posY; // inicializa o atributo posY
    }

    public void MoverParaBaixo() { posX += 1; }
    public void MoverParaDireita() { posY += 1; }
    public void Ampliar() { raio += 1; }
}
```

Modificadores de Acesso public e private

- Com nossa classe de exemplo Circulo, o que aconteceria se o atributo raio dos objetos fosse acessado a partir do programa principal?

```
var c1 = new Circulo(8, 20, 25);
Console.WriteLine(c1.raio);
```

readonly struct System.Int32
Represents a 32-bit signed integer.

CS0122: 'Circulo.raio' is inaccessible due to its protection level

A mensagem do compilador informa que o acesso não é permitido. Isso acontece porque em nosso exemplo os atributos foram definidos apenas pela indicação do tipo e do nome do atributo (int raio; int posx etc.). Atributos definidos dessa forma são privados, por padrão. Isso significa que só podem ser acessados pelo código dentro da própria classe (métodos e construtor), o que não é o caso do acesso neste exemplo.

Modificadores de Acesso public e private

- O modificador de acesso **private** aplicado a atributos e métodos restringe o acesso a esses itens apenas por código interno da classe.

```
class Circulo
{
    // Atributos
    int raio;
    int posX;
    int posY;

    // Método construtor com três parâmetros
    public Circulo(int raio, int posX, int posY)
    {
        this.raio = raio; // inicializa o atributo raio
        this.posX = posX; // inicializa o atributo posX
        this.posY = posY; // inicializa o atributo posY
    }

    // Exemplo de método acessando o atributo privado 'raio'
    public double CalcularArea() { return Math.PI * raio*raio; }
};
```

Essas definições são equivalente a:
private int raio;
private int posX;
private int posY;

```
var c1 = new Circulo(8, 20, 25);
Console.WriteLine(c1.raio); // Acesso não permitido aqui, pois raio é privado
```

Modificadores de Acesso public e private

- O modificador de acesso **public** permite que o atributo/método seja acessado de qualquer lugar, seja por código interno da classe ou por código externo, como neste exemplo.

```
class Circulo
{
    // Atributos
    public int raio;
    private int posX;
    private int posY;

    // Método construtor com três parâmetros
    public Circulo(int raio, int posX, int posY)
    {
```

```
var c1 = new Circulo(8, 20, 25);
Console.WriteLine(c1.raio); // permitido, pois raio é público
Console.WriteLine(c1.posX); // não permitido, pois posX é privado
```

Modificadores public e internal em Classes

- Classes definidas com **public** podem ser acessadas de **qualquer parte do projeto** e também por **outros projetos** (com as devidas referências).
- Quando nenhum modificador é usado na definição da classe, será considerado o modificador padrão **internal**.
- O modificador **internal** permite que a classe seja acessada apenas por código dentro do mesmo projeto (assembly), mas não por código de outros projetos.

Objeto como Parâmetro de Método

```
class Ponto2D
{
    public int x;
    public int y;

    public Ponto2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Calcula a distancia do ponto atual
    // até outro ponto fornecido como parâmetro
    public double DistanciaAte(Ponto2D p)
    {
        var difX = p.x - this.x;
        var difY = p.y - this.y;
        return Math.Sqrt(difX * difX + difY * difY);
    }
};
```

```
var p1 = new Ponto2D(0, 0);
var p2 = new Ponto2D(3, 4);

// Mostra a distancia de p1 a p2
Console.WriteLine(p1.DistanciaAte(p2));
```

Modificadores de Acesso public e private - Reforçando

Modificador **public**

- Concede o nível máximo de acesso.
- O atributo ou método declarado como public pode ser acessado de qualquer lugar (de dentro da própria classe ou de outras classes).
- Define a interface externa do objeto, ou seja, a maneira como os outros objetos devem interagir com ele.

Modificador **private**

- Concede o nível mínimo de acesso.
- O atributo ou método declarado como private só pode acessado de dentro da própria classe onde foi declarado.
- É o modificador padrão para atributos e métodos em C# (quando nenhum modificador é especificado, private é utilizado internamente)

Porque simplesmente não usar public em tudo?

- Há muitas situações práticas onde certos atributos e métodos são relevantes apenas no contexto interno da classe.
- Portanto, tais atributos e métodos podem ser declarados como privados, evitando a exposição direta ao código externo.
- Isso ajuda a proteger o estado interno do objeto, evitando modificações indesejadas ou inconsistentes.
- O modificador private também pode ajudar a esconder a complexidade da classe.
  - Detalhes de implementação não são expostos ao código externo.
  - Portanto, podem ser aprimorados com o tempo sem interferir na interface de acesso da classe.
- O conceito de encapsulamento é baseado na ideia de se criar classes com proteção de dados, controle de acesso e complexidade escondida.

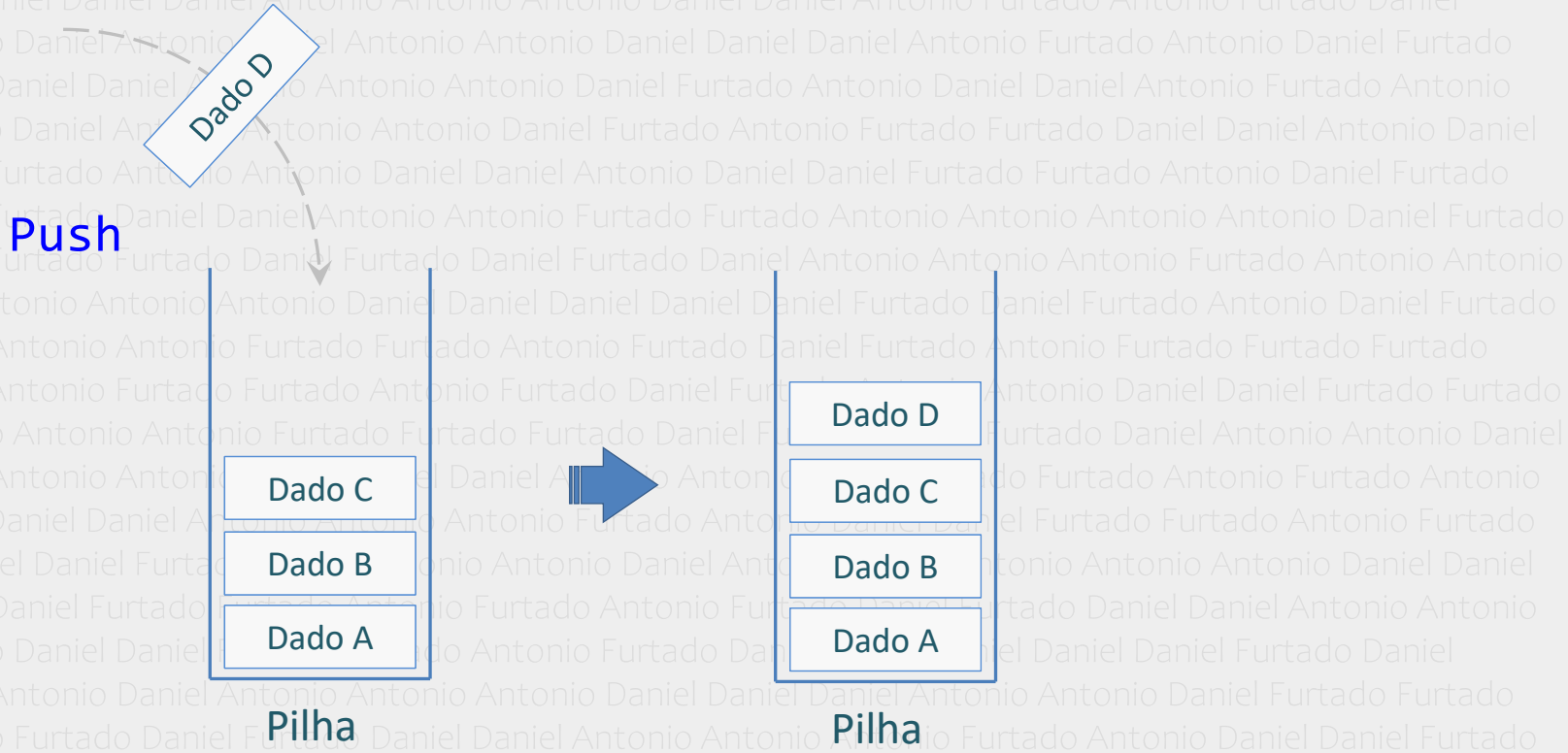
Encapsulamento

- O conceito de encapsulamento no contexto de classes e objetos pode ser melhor entendido fazendo analogia com um aparelho celular.
- O celular possui muitos componentes internos de alta complexidade que ficam escondidos do usuário (private).
  - Estão protegidos internamente.
- No entanto, alguns poucos elementos como o display e os botões de volume/liga/desliga são expostos (public) para que o usuário possa interagir com o aparelho (sem conhecer a sua complexidade interna).
- À medida em que o celular evolui, seus componentes internos tendem a ser aprimorados, mas a interface de interação com o usuário (botões, tela etc.) geralmente não se altera de forma significativa.
- A mesma ideia se aplica no contexto de classes e objetos em POO.

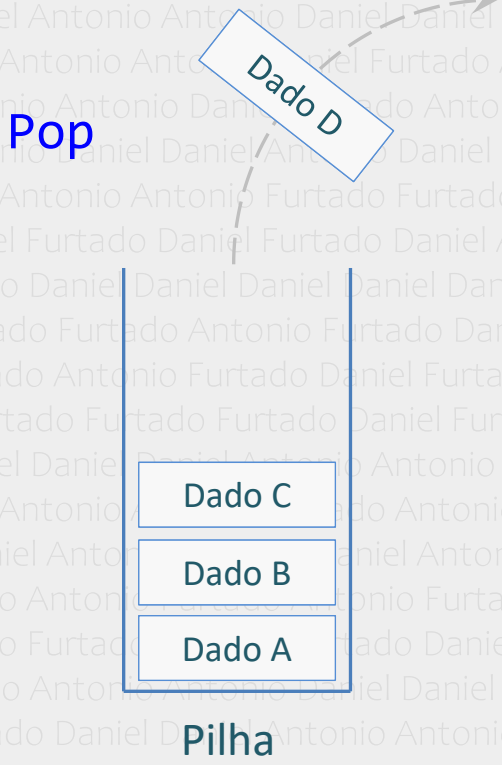
Exemplo de Encapsulamento - Estrutura Pilha

- Análogo a uma pilha de pratos em um restaurante.
- Dados podem ser adicionados e removidos de uma única extremidade, denominada **topo** da pilha.
- O último dado inserido na pilha será o primeiro a ser removido.
- **L**ast **I**n **F**irst **O**ut (LIFO) - último a entrar, primeiro a sair.
- Operações fundamentais:
  - **Push**: adiciona um elemento no topo da pilha
  - **Pop**: remove e retorna o elemento do topo da pilha

Exemplo de Encapsulamento - Estrutura Pilha



Exemplo de Encapsulamento - Estrutura Pilha



Exemplo de Encapsulamento - Estrutura Pilha

```
class Pilha
{
    private int[] itens;
    private int topo;

    public Pilha(int capacidade) {
        itens = new int[capacidade];
        topo = -1;
    }

    public void Push(int item)  {
        if (topo == itens.Length - 1)
            throw new Exception("Pilha cheia!");

        topo++;
        itens[topo] = item;
    }

    public int Pop() {
        if (topo == -1)
            throw new Exception("Pilha vazia!");

        var itemNoTopo = itens[topo];
        topo--;
        return itemNoTopo;
    }
}
```

```
// cria uma pilha para armazenar até 5 inteiros
var pilha1 = new Pilha(5);
pilha1.Push(1); // adiciona 1 na pilha
pilha1.Push(2); // adiciona 2 na pilha
pilha1.Push(3); // adiciona 3 na pilha

Console.WriteLine(pilha1.Pop()); // mostra 3
Console.WriteLine(pilha1.Pop()); // mostra 2
Console.WriteLine(pilha1.Pop()); // mostra 1
```

Repare que os atributos itens e topo foram definidos como privados, pois eles não podem ser expostos ao código externo. Se o usuário (código externo) pudesse acessar diretamente o array itens ou o atributo topo e modificar os valores, toda a lógica de funcionamento da pilha seria quebrada. O que interessa de fato para o código externo são os métodos Push e Pop. Esses de fato precisam ser definidos como public.

Exposição Direta de Atributos com public

- Na classe Circulo, poderíamos pensar em definir raio como público para permitir sua livre atualização por parte do usuário (código externo).
- Porém, essa abordagem não evitaria, por exemplo, que o usuário definisse um raio negativo, o que não existe na prática.

```
class Circulo
{
    // Atributos
    public int raio;
    private int posX;
    private int posY;

    // Método construtor com três parâmetros
    public Circulo(int raio, int posX, int posY)
    {
```

```
var c1 = new Circulo(8, 20, 25);
c1.raio = -2;
```

Exposição Indireta com *Getters* e *Setters*

- Nesse caso, uma alternativa melhor é definir raio como privado e criar métodos que ofereçam acesso indireto ao atributo.
- Tais métodos podem intermediar o acesso e impedir a atribuição de um valor nulo ou negativo.
- Esses métodos em POO são chamados de *getters* e *setters* (ou *acessadores*)
  - Possuem a responsabilidade de acessar e modificar os atributos.
- *Getters* e *setters* contribuem para reforçar o conceito de encapsulamento.
- Os nomes devem seguir o padrão PascalCase.
  - `GetRaio()`, `SetRaio()`, `GetPosicaoX()`, `SetPosicaoX()` etc.

Exposição Indireta com *Getters* e *Setters*

```
class Circulo
{
    private int raio;

    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
    }

    // Método do tipo getter para acessar o atributo raio
    public int GetRaio() { return raio; }

    // Método do tipo setter para modificar o atributo raio
    public void SetRaio(int raio)
    {
        if (raio > 0)
            this.raio = raio;
        else
            Console.WriteLine("O raio não pode ser 0 ou negativo");
    }
};
```

```
var c1 = new Circulo(8); // Cria um círculo com raio 8
c1.SetRaio(5); // Modifica o raio para 5
Console.WriteLine(c1.GetRaio()); // Mostra '5'
c1.SetRaio(-3); // Tenta definir um raio negativo
Console.WriteLine(c1.GetRaio()); // Mostra '5' novamente
```

Exposição Indireta com *Getters* e *Setters*

- Não é obrigatório definir *getters* e *setters* para todos os atributos privados.
- Se um atributo só é relevante para o código interno da classe, pode ser desnecessário criar um respectivo *getter* ou *setter*.
- Eventualmente, apenas um dos métodos precisa ser criado.
  - Por exemplo, para expor um atributo apenas para leitura, basta criar o respectivo *getter*.

Exposição Indireta com *Getters* e *Setters*

```
class Circulo
{
    private int raio;
    private int posX;
    private int posY;

    public Circulo(int raio, int posX, int posY)
    {
        this.raio = raio;
        this.posX = posX;
        this.posY = posY;
    }

    public int GetRaio() { return raio; }
    public void SetRaio(int raio)
    {
        if (raio > 0)
            this.raio = raio;
    }

    public int GetPosX() { return posX; }
    public int GetPosY() { return posY; }
    public void MoverParaDireita() { posX += 1; }
    public void MoverParaBaixo() { posY += 1; }
};
```

```
var c1 = new Circulo(5, 20, 30);
Console.WriteLine(c1.GetPosX()); // mostra 20
Console.WriteLine(c1.GetPosY()); // mostra 30
Console.WriteLine(c1.GetRaio()); // mostra 5

c1.SetRaio(10);
Console.WriteLine(c1.GetRaio()); // mostra 10

c1.SetRaio(-2);
Console.WriteLine(c1.GetRaio()); // mostra 10

c1.MoverParaDireita();
Console.WriteLine(c1.GetPosX()); // mostra 21
```

Exercício

- Defina uma classe para representar veículos. Deve haver atributos privados para armazenar o modelo, o ano e a velocidade do veículo:
  - O construtor deve permitir que o usuário informe o modelo e o ano no momento de criar os objetos;
  - A velocidade inicial de todo veículo deve ser 0;
  - O modelo e o ano do veículo devem ser expostos ao código externo apenas para leitura (GetModelo/GetAno);
  - A velocidade deve ser exposta para leitura e atualização, porém não deve ser permitida uma velocidade negativa;
  - Em nenhum local do código interno da classe deve haver apresentação de mensagens ao usuário.
- Crie um "programa principal" ilustrando o uso da classe e dos métodos.

Trocando getters e setters por Propriedades

- Propriedades são um recurso do C# que visa oferecer a ideia central dos getters e setters, porém com facilidades de acesso adicionais.
- Elas combinam a simplicidade de acesso de um atributo com o controle e a lógica de um método.
  - `circulo1.raio = -5; // O atributo é mais simples de acessar.`
  - `circulo1.SetRaio(-5); // Mas o método é mais seguro.`
- Para o código externo, a propriedade parece ser um simples atributo público. Porém, internamente, ela é implementada de forma similar aos métodos get/set.
- Ao invés de ter dois métodos distintos (GetRaio e SetRaio), a propriedade oferece as duas funcionalidades com um único identificador de acesso.
- Ao usá-la em um contexto de atribuição, internamente será chamado o "set". Ao usá-la em contexto de leitura, será chamado o "get".

Definindo Propriedades - Sintaxe Geral

```
<modificadorDeAcesso> tipoDeRetorno NomeDaPropriedade
{
    get
    {
        // código que é executado quando o usuário ACESSA
        // a propriedade (leitura).
        // Precisa retornar um valor.
    }
    set
    {
        // código que é executado quando o usuário ATRIBUI
        // um valor à propriedade (escrita).
        // Aqui dentro, a palavra-chave value representa
        // o valor sendo atribuido.
    }
}
```

Definindo Propriedades - Exemplo

```
class Circulo
{
    private int raio; // atributo raio (com r minúsculo)

    public Circulo(int raio) { this.raio = raio > 0 ? raio : 1; } // construtor

    public int Raio // Define a propriedade Raio (com R maiúsculo)
    {
        get { return raio; } // Retorna o valor do atributo raio
        set
        {
            // value é uma palavra reservada que representa o valor sendo atribuído
            if (value > 0)
                raio = value; // Altera o atributo raio usando o valor atribuído
            else
                Console.WriteLine("O raio não pode ser 0 ou negativo");
        }
    }
}
```

```
var c1 = new Circulo(8); // Cria um círculo com raio 8
c1.Raio = 5; // Modifica o raio para 5 usando a propriedade (atribuição)
c1.Raio = -3; // Tenta alterar o raio para -3, mas a propriedade não aceitará.
Console.WriteLine(c1.Raio); // Acessa a propriedade (leitura) e apresenta '5'
```

Propriedades Computadas – Somente Leitura

- Outra vantagem das propriedades é a possibilidade de criá-las como valores derivados de cálculos, sem que haja um atributo de apoio por trás.
- Neste caso, a propriedade será de apenas leitura e não deve ter o bloco set.

```
class Circulo
{
    private int raio;

    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
    }

    // Propriedade computada, somente leitura, sem atributo de apoio.
    public double Area
    {
        get { return Math.PI * raio*raio; }
    }
};
```

```
var c1 = new Circulo(5); // Cria um círculo com raio 5
Console.WriteLine(c1.Area); // Mostra a área do circulo
```

Exercício

- Defina uma classe para modelar um Retangulo conforme especificações a seguir:
  - Deve haver atributos privados para armazenar a base e a altura do retângulo (utilize o tipo double);
  - O construtor deve permitir a criação de novos retângulos informando a respectiva base e altura;
  - Crie as propriedades Base e Altura para fornecer acesso indireto aos atributos base e altura do retângulo. As propriedades não devem permitir valores nulos ou negativos;
  - Crie uma propriedade computada que retorne a área do retângulo.
- Crie um "programa principal" ilustrando o uso da classe e das propriedades.

Propriedades Computadas com o Operador =>

- C# disponibiliza o operador seta => para situações onde o corpo do método pode ser substituído por uma única expressão (operador de corpo de expressão).
- Permite que a declaração seja compacta, clara e prática.
- Propriedades computadas, por exemplo, podem ser declaradas em uma única linha.

```
class Circulo
{
    private int raio;

    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
    }

    // Propriedade computada, somente leitura, com ⇒
    public double Area ⇒ Math.PI * raio * raio;
};
```

Métodos e Construtores com o Operador =>

Se o corpo do método ou construtor possui uma única sentença, ele também pode ser escrito de forma concisa com o operador seta =>.

```
class Circulo
{
    private int raio;

    // Construtor com operador 'seta'.
    public Circulo(int raio) ⇒ this.raio = raio > 0 ? raio : 1;

    // Propriedade com operador 'seta'.
    public double Area ⇒ Math.PI * raio * raio;

    // Método com operador 'seta'.
    public void Descrever() ⇒ Console.Write($"Círculo de raio: {raio}");
};
```

Get e Set de Propriedades com o Operador =>

A mesma ideia se aplica aos *acessadores* Get/Set de propriedades, que também podem ser escritos de forma concisa e compacta.

```
class Circulo
{
    private int raio;

    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
    }

    // Propriedade raio definida com ⇒
    // O atributo raio receberá o valor fornecido pelo usuário
    // na atribuição caso esse valor seja positivo. Caso contrario,
    // o atributo receberá o seu próprio valor (não mudará).
    public int Raio
    {
        get ⇒ raio;
        set ⇒ raio = value > 0 ? value : raio;
    }

    // Propriedade computada, somente leitura, com ⇒
    public double Area ⇒ Math.PI * raio * raio;
};
```

```
get { return raio; }
```

Foi substituído por:

```
get => raio;
```

```
set
{
   if (value > 0)
      raio = value
}
```

Foi substituído por:

```
set => raio = value > 0 ? value : raio;
```

Propriedade Autoimplementada

- Propriedades que acessam diretamente os respectivos atributos sem processamentos adicionais no get/set (como a seguir) podem ser declaradas de maneira simplificada.

```
public int Raio {
    get { return this.raio; }
    set { this.raio = value; }
}
```

- Nesses casos, ao utilizar a sintaxe a seguir, não é necessário definir explicitamente o respectivo campo privado (raio), pois o compilador criará automaticamente um campo anônimo para esse fim.

```
public int Raio { get; set; }
```

OBS: Esse tipo de propriedade, a princípio, pode parecer desnecessário, uma vez que poderíamos pensar em criar um simples atributo público para ter um efeito similar. Entretanto, a criação da propriedade trás mais flexibilidade e segurança, pois é possível que no futuro a sua implementação seja alterada (incluindo validações e processamentos extras, por exemplo). Com a propriedade, mesmo que haja mudanças na implementação, é possível que seu acesso permaneça inalterado.

Conteúdo Complementar

Propriedade Autoimplementada - Exemplo

```
class Circulo
{
    // Atributo privado.
    private int raio;

    // Propriedades autoimplementadas. Os atributos 'por trás'
    // são gerados automaticamente pelo compilador.
    public int PosicaoX { get; set; }
    public int PosicaoY { get; set; }
    public string? Cor { get; set; }

    // Construtor
    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
    }

    // Propriedade convencional.
    public int Raio
    {
        get ⇒ raio;
        set ⇒ raio = value > 0 ? value : raio;
    }
};
```

Conteúdo Complementar

Sintaxe Moderna de Inicialização de Propriedades

- Ao criar o construtor da classe, definimos quais atributos do objeto são obrigatórios para sua instanciação
  - Por meio dos parâmetros do construtor.
- Entretanto, é possível que haja atributos opcionais, que o usuário pode ou não precisar fornecer, dependendo do contexto.
- Nessas situações não é necessário criar um grande número de construtores com variações de parâmetros, pois há uma sintaxe para fácil inicialização do objeto (denominada inicializador de objeto):

```
var objeto = new MinhaClasse
{
   NomeDaPropriedade1 = valor1;
   NomeDaPropriedade2 = valor2;
   NomeDaPropriedade3 = valor3;
   ...
}
```

Conteúdo Complementar

Sintaxe Moderna de Inicialização de Propriedades

```
// Cria um novo círculo de raio 10 e inicializa
// as propriedades PosicaoX, PosicaoY e Cor
var circulo1 = new Circulo(10)
{
    PosicaoX = 50,
    PosicaoY = 30,
    Cor = "Vermelho"
};

// Cria um novo círculo de raio 20 e inicializa
// apenas as propriedades PosicaoX e PosicaoY
var circulo2 = new Circulo(20)
{
    PosicaoX = 100,
    PosicaoY = 150
};
```

Conteúdo Complementar

Exercício

- Defina uma classe para modelar um veículo conforme detalhado a seguir:
  - Deve haver atributos privados para armazenar o modelo, o ano e a velocidade do veículo:
  - O construtor deve permitir que o usuário informe o modelo e o ano no momento de criar os objetos;
  - A velocidade inicial de todo veículo deve ser 0;
  - O modelo e o ano do veículo devem ser expostos ao código externo, somente para leitura, definindo as propriedades Modelo e Ano;
  - Crie a propriedade Velocidade para expor o atributo velocidade para leitura e atualização. A propriedade não deve permitir um valor negativo;
- Crie um "programa principal" ilustrando o uso da classe e das propriedades.

Membros de Instância vs Membros Estáticos

- Todos os atributos, propriedades e métodos criados nos exemplos anteriores estão vinculados ao objeto em si.
- Cada objeto círculo criado a partir da classe Círculo possui o seu próprio raio, sua própria posição, sua própria cor.
- Ao escrever c1.Ampliar(), c1.MoverParaBaixo(), c1.CalcularArea() estamos chamando métodos que irão atuar no objeto em questão c1.
  - Para isso, precisamos criar primeiramente o objeto (c1 = new Circulo(...));
- Membros (atributos, métodos, propriedades etc.) que estão vinculados aos objetos são denominados membros de instância.
  - Atributo de instância, método de instância, propriedade de instância etc.
- Porém, há situações onde é mais adequado criar membros que estejam associados à classe em si, e não a cada objeto individualmente.
  - São denominados membros de classe ou membros estáticos.

Membros Estáticos

- A sentença Console.WriteLine() utilizada para mostrar mensagens ao usuário ilustra o conceito de membro estático.
- WriteLine é um método estático da classe Console do framework .NET.
- Repare que o método WriteLine é chamado a partir do nome da classe.
  - Não é necessário criar um objeto da classe Console para posteriormente chamar o método a partir do objeto (o que dificultaria o uso).
- Portanto, membros estáticos são referenciados usando o nome da classe ao invés do nome do objeto.

Membros Estáticos

- O método Math.Sqrt(x), para cálculo da raiz quadrada de x, é outro exemplo de método estático;
  - Não é necessário criar uma instância da classe Math para poder chamar o método Sqrt.
- A classe Math do framework .NET tem muitos outros métodos e propriedades estáticos:
  - Math.PI – retorna o valor da constante matemática π
  - Math.Max(a, b) – calcula o maior valor entre a e b
  - Math.Round(x) – arredonda x para o inteiro mais próximo
  - Math.Abs(x) – calcula o valor absoluto (módulo) de x
  - Math.Sin(x) – calcula o seno do ângulo x (medido em radianos)

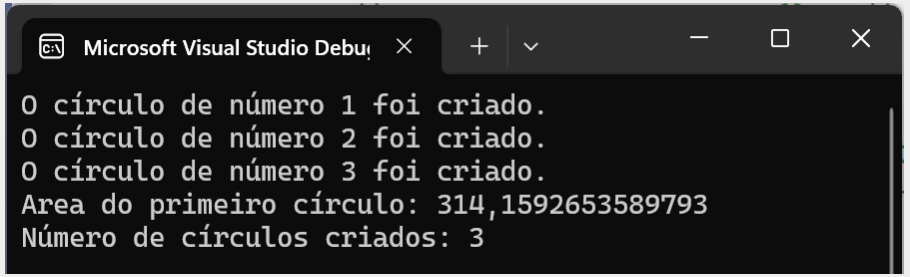
Criando Atributos Estáticos

- Suponha que seja necessário gerenciar a quantidade de objetos do tipo Circulo que são criados pela aplicação.
- Uma solução é criar um atributo estático na classe Circulo para manter a quantidade de objetos instanciados.
  - Esse atributo deve ser inicializado em 0 e incrementado à cada novo objeto instanciado.
- Para definir um atributo, método ou propriedade como estático, basta utilizar a palavra reservada static antes do tipo:
  - `private static int count;`

Criando Atributos Estáticos

```
class Circulo
{
    private int raio; // Atributo de instância
    private static int count = 0; // Atributo de classe (estático)

    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
        count++; // incrementa o atributo de classe
        Console.WriteLine($"O círculo de número {count} foi criado.");
    }

    // Propriedade de instância
    public double Area ⇒ Math.PI * raio * raio;

    // Propriedade de classe (estática)
    public static int Count ⇒ count;
};
```

```
var c1 = new Circulo(10); // Cria círculo de raio 10
var c2 = new Circulo(15); // Cria círculo de raio 15
var c3 = new Circulo(20); // Cria círculo de raio 20

Console.WriteLine($"Área do primeiro círculo: {c1.Area}");
Console.WriteLine($"Número de círculos criados: {Circulo.Count}");
```



Observações sobre Atributos Estáticos

- Pertence à classe, não ao objeto.
  - Existe independentemente das instâncias criadas (objetos).
- Cópia única e compartilhada.
  - Existe apenas uma cópia do atributo estático na memória, independentemente do número de objetos criados.
- Acesso externo pelo nome da classe (e não pelo nome do objeto).
  - Correto: Circulo.atributo
  - Incorreto: c1.atributo
- Ciclo de vida independente.
  - Inicializado quando a classe é utilizada pela primeira vez na aplicação.

Métodos Estáticos

- Executam ação no contexto da classe, não dos objetos.
  - Não podem alterar o estado dos objetos individualmente.
- Podem acessar apenas membros estáticos.
  - Não pode acessar atributos e propriedades de instância.
  - Não pode chamar métodos de instância.
- Podem ser chamados a partir de métodos estáticos e não estáticos.
- O acesso externo deve ser feito sempre pelo nome da classe.
  - Correto: Circulo.MetodoEstatico()
  - Incorreto: c1.MetodoEstatico()

Exemplo de Método Estático

```
class Circulo
{
    private int raio; // Atributo de instância
    private static int count = 0; // Atributo de classe (estático)

    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
        count++; // incrementa o atributo de classe a cada objeto criado
    }

    // Método estático para exibir o número de instâncias criadas
    public static void MostrarQuantidade()
    {
        Console.WriteLine($"Número de círculos criados: {count}");
    }
};
```

```
var c1 = new Circulo(10); // Cria círculo de raio 10
var c2 = new Circulo(15); // Cria círculo de raio 15
var c3 = new Circulo(20); // Cria círculo de raio 20

Circulo.MostrarQuantidade(); // Mostra "Número de círculos criados: 3"
```

Métodos Estáticos – Uso Inadequado

```
class Circulo
{
    private int raio; // Atributo de instância
    private static int count = 0; // Atributo de classe (estático)

    public Circulo(int raio)
    {
        this.raio = raio > 0 ? raio : 1;
        count++;
    }

    // Método estático não pode acessar atributo de instância
    public static void Calcular()
    {
        double area = Math.PI * raio * raio; // ERRO - raio é atributo de instância
    }
};
```

Neste exemplo o método estático Calcular tenta acessar o atributo de instância raio, mas o compilador aponta o erro e destaca a variável raio com sublinhado vermelho.

Classes Estáticas

- Em algumas situações pode ser útil agrupar um conjunto de métodos (ou propriedades) correlacionados dentro de uma classe que tem como único propósito servir de container para o agrupamento (e não como molde para criação de novos objetos);
- Para esses casos pode-se criar uma classe estática;
- Uma classe estática deve possuir apenas membros estáticos
  - Não pode ter atributos, propriedades, métodos de instância ou construtor
- Uma classe estática não pode ser instanciada.
  - Portanto, não é permitido criar objetos da classe.
  - O uso deve ser feito sempre pelo nome da própria classe.
- Para definir uma classe estática basta utilizar a palavra reservada static:
  - static class MinhaClasseEstatica { ... }

Classes Estáticas - Exemplo

```
static class AreaFiguras
{
    public static double AreaQuadrado(double lado) => lado * lado;

    public static double AreaRetangulo(double b, double h) => b * h;

    public static double AreaTriangulo(double b, double h) => (b * h)/2;

    public static double AreaCirculo(double raio) => Math.PI * (raio*raio);
}
```

```
Console.WriteLine($"Area do quadrado de lado 3: {AreaFiguras.AreaQuadrado(3)}");
Console.WriteLine($"Area do retângulo de lados 3 e 4: {AreaFiguras.AreaRetangulo(3,4)}");
Console.WriteLine($"Area do triângulo de base 3 e altura 4: {AreaFiguras.AreaTriangulo(3, 4)}");
Console.WriteLine($"Area do círculo de raio 3: {AreaFiguras.AreaCirculo(3)}");
```

Classes Estáticas - Exemplo

```
class Cor
{
    private int r;
    private int g;
    private int b;

    public Cor(int r, int g, int b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public void Descrever()
    {
        Console.WriteLine($"Cor: R={r},G={g},B={b}");
    }
}
```

Classe para representação de cores no formato RGB (as cores são representadas por uma combinação de vermelho (Red), verde (Green) e azul (Blue)).

```
static class CoresComuns
{
    public static Cor Vermelho => new Cor(255, 0, 0);
    public static Cor Verde => new Cor(0, 255, 0);
    public static Cor Azul => new Cor(0, 0, 255);
    public static Cor Amarelo => new Cor(255, 255, 0);
    public static Cor Magenta => new Cor(255, 0, 255);
    public static Cor Ciano => new Cor(0, 255, 255);
    public static Cor Branco => new Cor(255, 255, 255);
    public static Cor Preto => new Cor(0, 0, 0);
    public static Cor Laranja => new Cor(255, 165, 0);
    public static Cor Roxo => new Cor(128, 0, 128);
}
```

Classe estática definindo as cores mais comuns como propriedades estáticas para facilitar o uso.

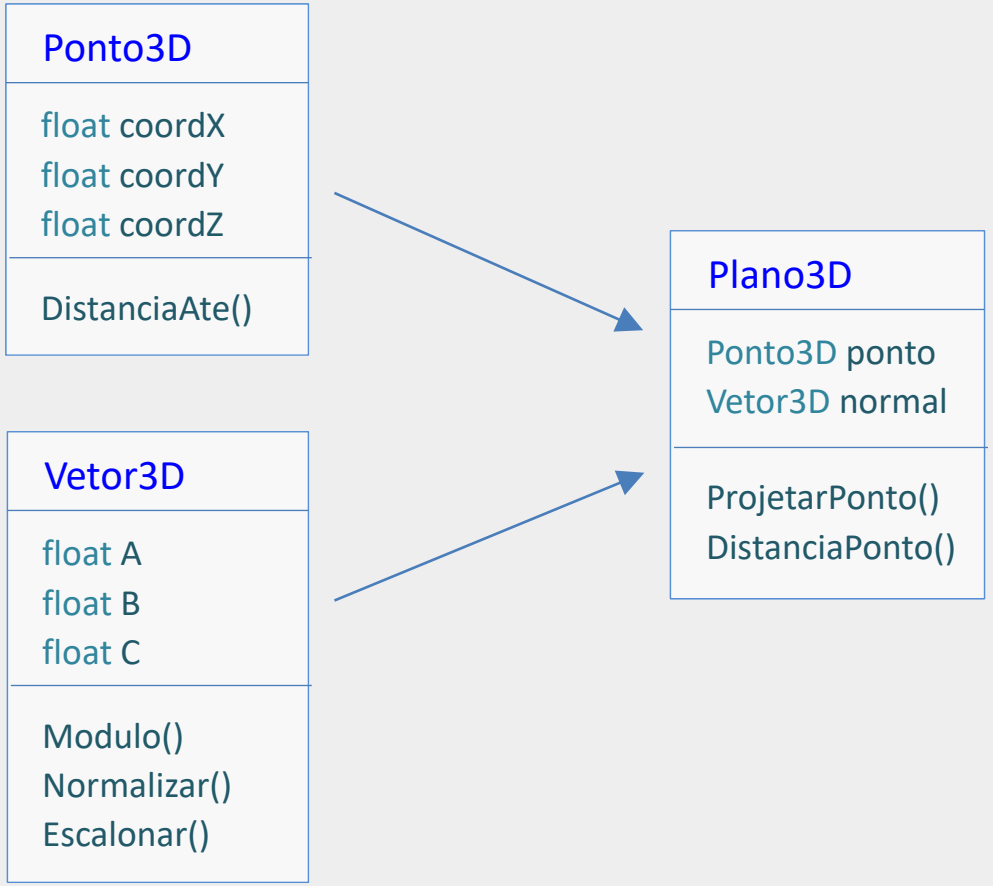
Classes Estáticas - Exemplo

```
var marrom = new Cor(165, 42, 42);
var cinza = new Cor(128, 128, 128);
var vermelho = CoresComuns.Vermelho;
var verde = CoresComuns.Verde;

cinza.Descrever(); // Mostra "Cor: R=128,G=128,B=128"
vermelho.Descrever(); // Mostra: "Cor: R=255,G=0,B=0"
```

Conceito de Agregação

- Novos objetos podem ser criados a partir da agregação de objetos existentes.
- Por exemplo, um objeto do tipo Computador poderia ser modelado como uma composição de objetos menores correspondentes a seus componentes (Mouse, Teclado, Monitor e Gabinete).
- De forma similar, um objeto do tipo Circulo pode conter um objeto do tipo Cor.
- Matematicamente, um plano é definido por um ponto e um vetor normal. Dessa forma, um objeto do tipo Plano3D pode conter um objeto do tipo Ponto3D e um objeto do tipo Vetor3D.

Exemplo de Agregação



```
class Ponto3D
{
    public float x;
    public float y;
    public float z;

    public Ponto3D(float x, float y, float z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

```
class Vetor3D
{
    public float a;
    public float b;
    public float c;

    public Vetor3D(float a, float b, float c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public float Modulo() => (float)Math.Sqrt(a*a + b*b + c*c);
}
```

```
class Plano3D
{
    public Ponto3D p;
    public Vetor3D vetorNormal;

    public Plano3D(Ponto3D p, Vetor3D vetorNormal)
    {
        this.p = p;
        this.vetorNormal = vetorNormal;
    }
}
```

```
var p = new Ponto3D(0, 0, 0);
var vetorNormal = new Vetor3D(1, 1, 1);
var plano1 = new Plano3D(p, vetorNormal);
var plano2 = new Plano3D(
    new Ponto3D(1,2,3), new Vetor3D(4,5,6));
```

Programa principal

```
class Cor
{
    private int r;
    private int g;
    private int b;

    public Cor(int r, int g, int b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public void Descrever()
    {
        Console.WriteLine($"Cor: R={r},G={g},B={b}");
    }
}
```

```
class Circulo
{
    private int raio; // Atributo de instância
    private Cor cor;

    public Circulo(int raio, Cor cor)
    {
        this.raio = raio > 0 ? raio : 1;
        this.cor = cor;
    }

    public double Area => Math.PI * raio * raio;

    public void Descrever()
    {
        Console.Write($"Raio: {raio} ");
        cor.Descrever();
    }
};
```

```
// Cria círculo cinza de raio 10
var c1 = new Circulo(10, new Cor(128, 128, 128));

// Cria círculo vermelho de raio 15
var c2 = new Circulo(15, CoresComuns.Vermelho);

c1.Descrever(); // Mostra "Raio: 10 Cor: R=128,G=128,B=128"
c2.Descrever(); // Mostra: "Raio: 15 Cor: R=255,G=0,B=0"
```

Exercício

- Defina uma classe para modelar endereços conforme descrição a seguir:
  - Deve haver atributos privados para armazenar o cep, a rua e a cidade;
  - O construtor deve permitir a criação de novos endereços informando o cep, a rua e a cidade;
  - Crie um método para exibição dos dados do objeto.
- Defina uma classe para modelar clientes conforme descrição a seguir:
  - Deve haver atributos privados do tipo string para armazenar o CPF e o nome do cliente;
  - Deve haver atributos privados para armazenar o endereço residencial do cliente e o endereço de trabalho. Utilize a classe Endereco definida anteriormente;
  - Deve haver um construtor que permita a criação de novos objetos informando o CPF, o nome e os endereços (residencial e trabalho);
  - Crie um método para exibição dos dados do cliente.
- Crie um "programa principal" ilustrando o uso das classes. Crie dois objetos da classe Cliente: um com endereços distintos e outro com endereços iguais (o cliente trabalha onde mora).

Variáveis e Referências para Objetos

- Cada objeto instanciado ocupa espaço próprio em memória, com seus próprios valores para os atributos de instância.
- A variável que nomeia o objeto armazena, na verdade, apenas uma referência para à região de memória contendo os dados do objeto em si.
- Portanto, ao copiar a variável de um objeto para outra, estamos copiando apenas a referência.
  - A região de memória ocupada pelo objeto em si não é copiada.
  - Portanto, o objeto em si não é copiado.

OBS: caso seja necessário realizar uma copia de fato de todo o objeto, o programador precisará implementar ações específicas para garantir uma cópia profunda de todos os dados do objeto (o objeto precisa ser clonado).

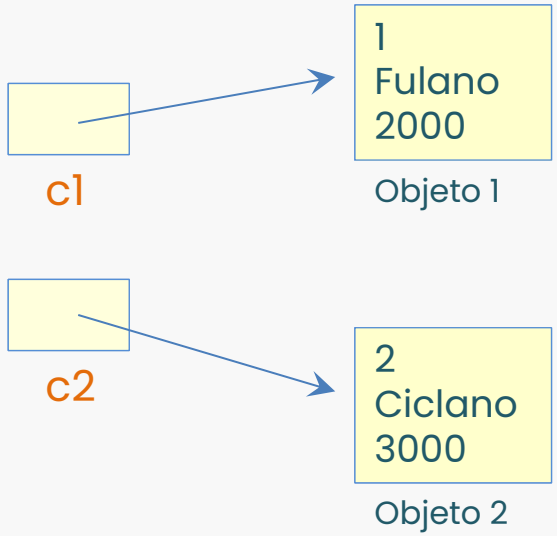
Variáveis e Referências para Objetos

Código

```
class Cliente
{
    private int codigo;
    private string nome;
    public decimal salario;

    public Cliente(int codigo, string nome, decimal salario)
    {
        this.codigo = codigo;
        this.nome = nome;
        this.salario = salario;
    }
}
```

```
var c1 = new Cliente(1, "Fulano", 2000);
var c2 = new Cliente(2, "Ciclano", 3000);
```

Memória

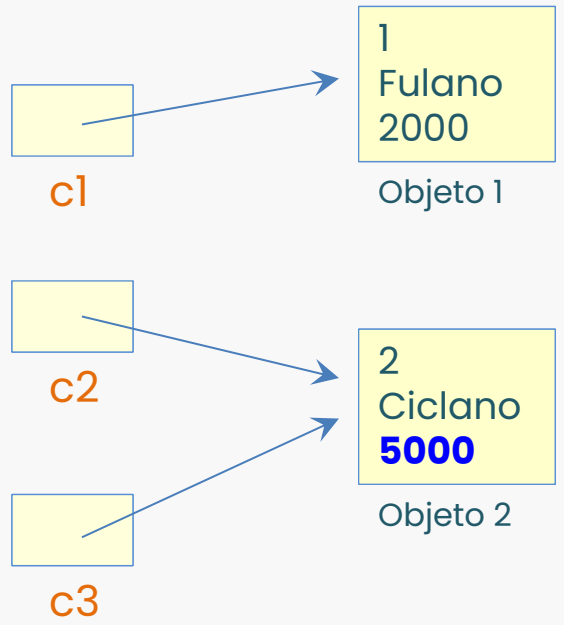


Variáveis e Referências para Objetos

Código

```
// c1 e c2 são referências para objetos distintos, cada um ocupando seu
// próprio espaço em memória.
var c1 = new Cliente(1, "Fulano", 2000);
var c2 = new Cliente(2, "Ciclano", 3000);

// A referência para o segundo objeto é copiada para a variável c3.
// Agora, c2 e c3 "apontam" para o mesmo objeto em memória.
var c3 = c2;
c3.salario = 5000; // Altera o salário do segundo cliente.

Console.Write(c1.salario);  // Mostra 2000.
Console.Write(c2.salario);  // Mostra 5000.
Console.Write(c3.salario);  // Mostra 5000, pois é o mesmo objeto.
```

Memória

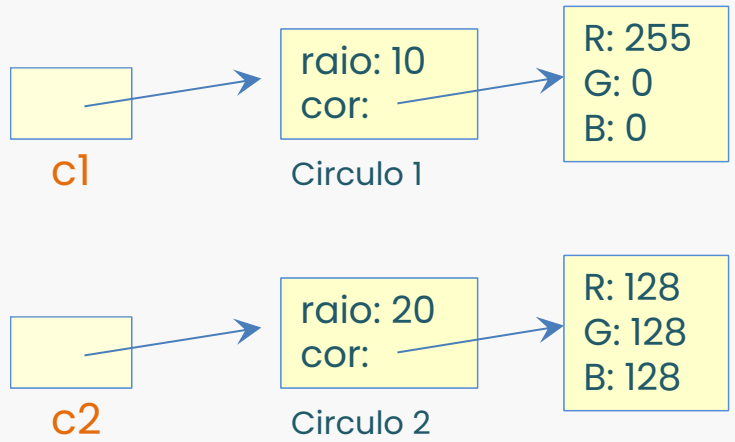


Variáveis e Referências para Objetos

Código

```
var c1 = new Circulo(10, CoresComuns.Vermelho);
var c2 = new Circulo(20, new Cor(128, 128, 128);
```

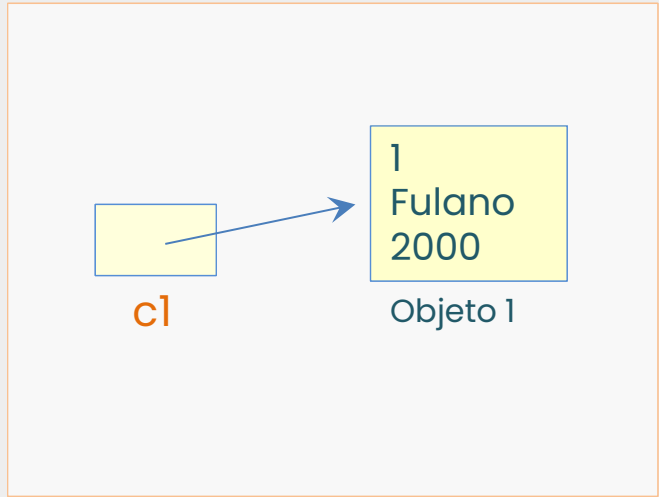
Memória



Variáveis e Referências para Objetos

- Depois de criar o objeto referenciado por c1 (figura ao lado) o que acontece se o programador atribuir null a c1? A região de memória continuará ocupada?
- E se o método onde o objeto foi criado finalizar? A região de memória ocupada pelo objeto será liberada?
- Em linguagens como C e C++, o programador precisa gerenciar essas situações manualmente:
  - Quando o objeto não é mais necessário, é preciso chamar uma função para liberar a respectiva região de memória.
- Em linguagens modernas como Java e C# há um mecanismo automático para liberação dessa memória:
  - O Garbage Collector (GC).

Memória




```
var c1 = new Cliente(1, "Fulano", 2000);
```

Garbage Collector

- O Garbage Collector (GC) é um processo automático que roda em segundo plano na aplicação para liberar a memória que não é mais utilizada.
- Os objetos são normalmente criados em uma região de memória gerenciada pelo GC: a heap genenciada.
  - Quando eles não são mais referenciados na aplicação, o GC os trata como "lixo" e os remove automaticamente da memória, liberando o espaço.
- Entretanto, o GC não "sabe" como fechar um arquivo aberto no sistema operacional, como encerrar uma conexão com o banco de dados ou como liberar um recurso gráfico.
  - Esses são os recursos não gerenciados (precisam de liberação manual).
  - Uma forma de liberar esses recursos é por meio do método destrutor.

Destrutores

- O destrutor é um método da classe que é chamado automaticamente pelo GC quando o objeto está prestes a ser coletado (memória liberada).
- Portanto, é a última oportunidade que o objeto tem para liberar recursos não gerenciados criados por ele.
- Como o destrutor é chamado pelo GC, o desenvolvedor não tem controle de quando ele será executado e se ele será executado.
  - A frequência de atuação do GC depende de vários fatores como da quantidade de memória disponível, da ociosidade da aplicação etc.

OBS 1: o desenvolvedor pode sugerir uma coleta ao GC chamando GC.Collect(). Porém, na prática, não há garantia de que o GC será de fato chamado naquele instante (no .NET moderno) e são raras as situações onde essa chamada é recomendada.

OBS 2: uma maneira melhor de liberar esse tipo de recurso é por meio da declaração using em conjunto com o método Dispose() e da interface IDisposable. Esse assunto será tratado mais adiante no curso.

Destrutores

- Em C# o método destrutor é criado com uma sintaxe especial:
  - Não possui tipo de retorno;
  - Tem o mesmo nome da classe;
  - Não tem parâmetros;
  - Tem o caractere ~ antes do nome do método.
- Exemplo:

```
class MinhaClasse
{
    ~MinhaClasse() // método destrutor
    {
        // Código para liberar recursos não gerenciados
    }
}
```

Registrando Mensagens de Log em Arquivo

```
public class Logger
{
    private StreamWriter arquivo;

    public Logger(string caminhoArquivo)
    {
        // Cria/abre arquivo para adicionar texto
        arquivo = new StreamWriter(caminhoArquivo, append: true);
    }

    public void RegistrarLog(string mensagem)
    {
        arquivo.WriteLine($"{DateTime.Now}: {mensagem}");
    }

    // Método para fechar o arquivo de logs
    public void Close() => arquivo.Close();

    ~Logger()
    {
        // Se o usuário esquecer de chamar o método Close(),
        // o destrutor ainda poderá fazê-lo.
        Close();
    }
}
```

```
// Cria um objeto para registrar logs em arquivo
var logger = new Logger("logs.txt");

// Cria ponto e registra no arquivo de logs
var p = new Ponto3D(0, 0, 0);
logger.RegistrarLog("Ponto criado.");

// Cria vetor e registra no arquivo de logs
var vetorNormal = new Vetor3D(1, 1, 1);
logger.RegistrarLog("Vetor normal criado.");

// Cria plano e registra no arquivo de logs
var plano1 = new Plano3D(p, vetorNormal);
logger.RegistrarLog("Primeiro plano criado.");

// Fecha o arquivo de logs
logger.Close();
```

Execução de Código C# no VS Code

1. No VS Code, instale a extensão **C# Dev Kit** da Microsoft;
2. Crie uma pasta para o projeto (ex. D:\Temp\Trab7);
3. Abra o terminal do VS Code: menu **Terminal** → **Novo Terminal;**
4. No terminal, navegue até a pasta criada digitando: **cd** D:\Temp\Trab7;
5. Digite **dotnet new console** para criar um **novo projeto** na pasta;
6. Digite "**code .**" para abrir o projeto em uma nova janela do VS Code;
7. Abra novamente o terminal e digital **dotnet run** para executar o código.
8. Para adicionar classes, acrescente um novo arquivo .cs no projeto utilizando o painel **Explorer** do VS Code.

Namespaces

- Console.WriteLine, Math.Sqrt, StreamWriter fazem parte da **biblioteca de classes** do framework .NET.
- Essa biblioteca disponibiliza **milhares** de classes (e outras estruturas) que fornecem inúmeras funcionalidades às aplicações como acesso à banco de dados, manipulação de strings, etc.
- Namespaces organizam essas classes em subgrupos e trazem dois benefícios principais:
  - Facilita a localização das classes dentro da imensa biblioteca de classes
  - Evita a obrigatoriedade de ter nomes distintos para todas as classes

Exemplos de Namespaces

- As classes Console e Math, que disponibilizam os métodos WriteLine e Sqrt, estão dentro do namespace System;
- Classes para manipulação de arquivos estão organizadas no namespace System.IO. Exemplos:
  - System.IO.File – classe com métodos para manipulação de arquivos
  - System.IO.Directory – classe com métodos para manipulação de pastas
- Casses relacionadas às redes estão sob o namespace System.Net como:
  - System.Net.Cookie – fornece métodos para manipulação de cookies
  - System.Net.WebRequest – fornece métodos para requisições HTTP
- Repare que namespaces podem conter outros namespaces, além das classes propriamente ditas. O namespace System, por exemplo, contém vários outros namespaces (como o namespace IO e o namespace Net)

Utilizando Namespaces

- Para utilizar as classes contidas nos namespaces há duas opções. Uma delas é utilizar o caminho completo incluindo o namespace e a classe:

```
var arquivoAberto = System.IO.File.OpenText("arquivo.txt");
var linhaTexto = arquivoAberto.ReadLine();
```

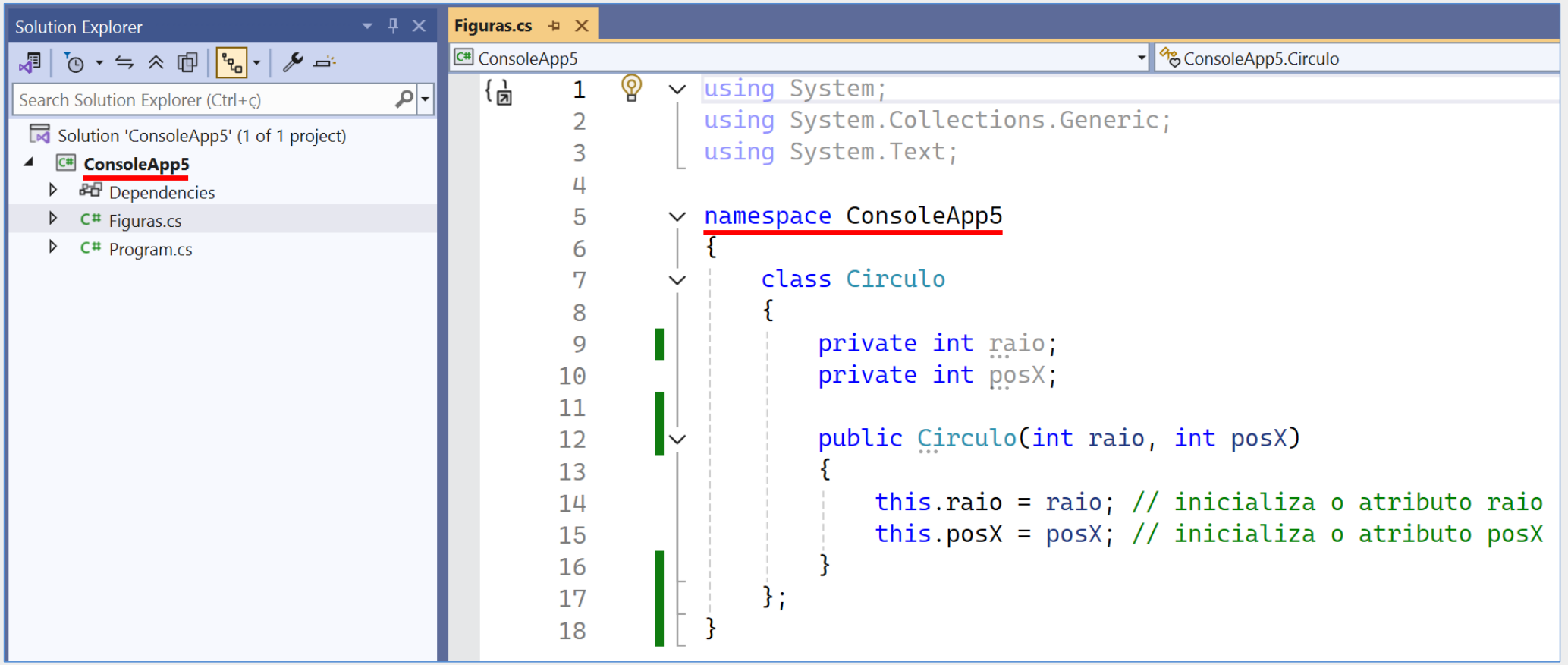
- A outra forma, mais prática, é utilizando a diretiva using NamespaceAlvo no início do arquivo .cs e referenciando a classe diretamente no código:

```
using System.IO;

var arquivoAberto = File.OpenText("arquivo.txt");
var linhaTexto = arquivoAberto.ReadLine();
```

using Globais

- É importante notar que o Visual Studio já inclui referências using automaticamente no código para namespaces mais comuns;
- Por esse motivo não é necessário escrever System.Console.WriteLine.
- Para visualizar esses namespaces no arquivo program.cs, basta clicar na pequena seta na região da margem, na primeira linha do arquivo:

Namespace Padrão no Projeto



Quando novos arquivos de código são adicionados ao projeto, o Visual Studio adiciona o novo código automaticamente dentro do **namespace padrão**, que tem o **mesmo nome do projeto**. Posteriormente, no arquivo Program.cs, é adicionado a diretiva using para esse namespace para permitir que tais classes sejam acessadas diretamente no código.

Definindo Namespaces

```
namespace Geo2D
{
    class Ponto  {
        public float x, y;
        public Ponto(float x, float y) { this.x = x; this.y = y; }
    }
    class Reta  {
        public float m, n;
        public Reta(float m, float n) { this.m = m; this.n = n; }
    }
}

namespace Geo3D
{
    class Ponto {
        public float x, y, z;
        public Ponto(float x, float y, float z)  {
            this.x = x; this.y = y; this.z = z;
        }
    }
    class Reta {
        public Ponto p1, p2;
        public Reta(Ponto p1, Ponto p2) {
            this.p1 = p1; this.p2 = p2;
        }
    }
}
```

Arquivo Geometria.cs

```
var pontoNoPlano1 = new Geo2D.Ponto(1, 2);
var retaNoPlano1 = new Geo2D.Reta(1, 5);

var p3D1 = new Geo3D.Ponto(1, 2, 3);
var p3D2 = new Geo3D.Ponto(4, 5, 6);
var retaNoEspaco = new Geo3D.Reta(p3D1, p3D2);
```

Ex1: As versões 2D e 3D das classes Ponto e Reta são acessadas usando o caminho completo incluindo o namespace (Geo2D ou Geo3D) e o nome da classe.

```
using Geo3D;

var p3D1 = new Ponto(1, 2, 3);
var p3D2 = new Ponto(4, 5, 6);
var retaNoEspaco = new Reta(p3D1, p3D2);
```

Ex2: Utilizando using Geo3D no início do arquivo program.cs, podemos referenciar as classes do respectivo namespace diretamente. Porém, utilizar using Geo3D e using Geo2D ao mesmo tempo causaria conflito de nomes, pois o Visual Studio não saberia distinguir qual das classes Ponto o desenvolvedor está referenciando.

Pacotes na Linguagem Java

- Na linguagem Java, existe um mecanismo de organização muito similar conhecido como Pacotes (package).
- Ambos os recursos compartilham o mesmo propósito de agrupar classes relacionadas, controlar a visibilidade e evitar conflitos de nomes.
- A principal diferença é que no Java deve haver uma correspondência obrigatória entre os pacotes definidos pelo usuário e as subpastas do projeto.
- As palavras-chave mudam, mas a lógica se mantém: namespace e using no C# equivalem, respectivamente, a package e import no Java.

Estrutura List

- Listas em C# permitem armazenar uma coleção de elementos do mesmo **tipo**.
- Diferentemente de arrays, listas possuem **tamanho dinâmico**, podendo aumentar ou diminuir para acomodar os elementos.
- Podem ser definidas utilizando List<Tipo> (ex: List<int>, List<string>, List<Ponto2D>)
- Disponível em System.Collections.Generic.
- Elementos podem ser acessados por índice número iniciando em zero.

```
// Define uma lista de strings
List<string> nomes = ["Ana", "Bob"];

// Adiciona elemento no final
nomes.Add("Alice");

// Adiciona elemento no início
nomes.Insert(0, "João");

// Remove o segundo elemento
nomes.RemoveAt(1);

// Percorrendo a lista com foreach
// Mostra 'João', 'Bob', 'Alice'
foreach (var nome in nomes)
    Console.WriteLine(nome);

// Percorrendo com for
for (int i = 0; i < nomes.Count; i++)
    Console.WriteLine(nomes[i]);
```

Estrutura Queue

- Para armazenar elementos seguindo a lógica de uma **fila**, pode-se utilizar a classe Queue.
- Com Queue, os elementos são inseridos no final da estrutura e removidos do início (**FIFO** – First In, First out).
- O método Enqueue adiciona um elemento no final da fila, enquanto Dequeue remove e retorna o primeiro da fila.
- A sintaxe de definição é similar à lista: Queue<string>, Queue<int>, etc.
- Queue também está disponível no namespace System.Collections.Generic.

```
// Define uma fila de strings vazia
Queue<string> filaAtendimento = [];

// Adiciona elementos no final
filaAtendimento.Enqueue("Alice");
filaAtendimento.Enqueue("Bob");
filaAtendimento.Enqueue("Carlos");

// Remove elemento do início (Alice)
var proximo = filaAtendimento.Dequeue();

// Mostra o primeiro da fila sem remover (Bob)
Console.WriteLine(filaAtendimento.Peek());

// Mostra os elementos da fila, sem remover
// (Bob, Carlos)
foreach (var nome in filaAtendimento)
    Console.WriteLine(nome);
```

Estrutura Stack

- Para armazenar elementos seguindo a lógica de uma **pilha**, pode-se utilizar a classe Stack.
- Com Stack, os elementos são inseridos e removidos de uma mesma extremidade, que representa o topo da pilha (**LIFO** – Last In, First out).
- O método Push adiciona um elemento no topo da pilha, enquanto Pop remove e retorna o elemento do topo.
- A sintaxe de definição é similar à lista: Stack<string>, Stack<int>, etc.
- Stack também está disponível no namespace System.Collections.Generic.

```
// Define uma pilha de strings vazia
Stack<string> pilhaNomes = [];

// Adiciona elementos no topo
pilhaNomes.Push("Alice");
pilhaNomes.Push("Bob");
pilhaNomes.Push("Carlos");

// Remove o elemento do topo (Carlos)
var topo = pilhaNomes.Pop();

// Mostra o topo da pilha sem remover (Bob)
Console.WriteLine(pilhaNomes.Peek());

// Mostra os elementos da pilha, começando
// pelo topo, sem remover (Bob, Alice)
foreach (var nome in pilhaNomes)
    Console.WriteLine(nome);
```

Exercício

- Defina uma classe estática contendo um método estático para verificar, utilizando uma pilha, se uma string corresponde a um palíndromo. Para simplificar, considere que a string terá sempre um número par de caracteres. A string deve ser percorrida uma vez.
- Adicione na classe um método estático para verificar se uma expressão matemática (string) tem seus parênteses devidamente abertos e fechados (balanceados). Utilize uma pilha. O método deve retornar true ou false indicando se a expressão está balanceada ou não, respectivamente.

Timers e Eventos

```
public class Alarme
{
    private System.Timers.Timer timer;

    public Alarme()
    {
        timer = new System.Timers.Timer();
    }

    public void Armar(int segundos)
    {
        timer.Stop(); // para se já estiver em andamento
        timer.Interval = segundos * 1000; // tempo do timer
        timer.Elapsed += TocarAlarme; // método a ser executado
        timer.AutoReset = false; // para não reiniciar
        timer.Start(); // inicia o timer
    }

    private void TocarAlarme(object? sender, ElapsedEventArgs e)
    {
        Console.WriteLine($"Alarme {timer.Interval/1000} segs!!");
        Console.Beep(800, 500); // frequencia e duracao do som
    }

    public void Desarmar() => timer.Stop();
}
```

```
var alarme1 = new Alarme();
var alarme2 = new Alarme();
var alarme3 = new Alarme();
alarme1.Armar(3);
alarme2.Armar(5);
alarme3.Armar(10);
Console.WriteLine("Alarmes definidos!");
Console.ReadLine();
```

Geração de Números Aleatórios e Thread.Sleep

- Números aleatórios podem ser gerados utilizando a classe Random.
- O método Next permite gerar o próximo número aleatório entre 0 e um valor limite passador por parâmetro.
- Console.SetCursorPosition permite posicionar o cursor em um local específico da janela.
- O método estático Sleep(t) da classe Thread suspende a execução do código por t milissegundos.

```
// cria um objeto para gerar números aleatórios
var random = new Random();

// oculta o cursor e mostra 'estrelas' no console
// até que alguma tecla seja pressionada
Console.CursorVisible = false;
while (!Console.KeyAvailable)
{
    Console.Clear(); // limpa a janela do console
    for (int i = 0; i < 100; i++)
    {
        // gera valores aleatórios para posicionar
        // o cursor dentro dos limites da janela
        var posX = random.Next(Console.WindowWidth);
        var posY = random.Next(Console.WindowHeight);

        // posiciona o cursor nas posições aleatórias
        // e apresenta o caracter '+' (estrela)
        Console.SetCursorPosition(posX, posY);
        Console.Write("+");
    }

    // Suspende a execução por 100 milisegundos
    Thread.Sleep(100);
}
Console.CursorVisible = false; // volta o cursor
```

Tratamento de Exceções

- Exceções são erros que podem ocorrer durante a execução do programa.
- Por exemplo, ao tentar acessar uma posição em um array além de seus limites, será lançada uma exceção do tipo IndexOutOfRangeException, que poderá finalizar abruptamente o programa.
- Outros exemplos de ações que lançam exceção:
  - Tentar abrir um arquivo já aberto ou inexistente;
  - Tentar se conectar ao um banco de dados inexistente ou indisponível;
  - Tentar realizar uma divisão por zero;
  - Tentar converter um texto com caracteres alfabéticos para número inteiro.
- O tratamento de exceções permite que o programa lide com esses erros de forma controlada, evitando a finalização abrupta.

Tratamento de Exceções

- Exceções podem ser tratadas utilizando um bloco try/catch.
- O código com potencial de produzir erro deve ser inserido dentro de um bloco try { ... }
- O tratamento dos eventuais erros gerados no bloco try { ... } deve ser feito utilizando um bloco catch { ... }, que deve ser inserido em seguida.
- Quando uma exceção é lançada (ocorrência do erro) por uma linha de código dentro do try, a execução é deslocada para o respectivo bloco catch (portanto, o restante do código dentro do try não será executado).
- Se houver a necessidade de executar algum código finalizador, ocorrendo ou não a exceção, pode-se acrescentar um bloco finally { ... } no final.
- As exceções são representadas em POO como objetos de classes específicas. Cada classe representa um tipo de exceção em particular como "arquivo não encontrado", "índice fora dos limites do array", etc.

Tratamento de Exceções – Exemplo 1

- Vários tipos de erros podem ocorrer ao tentar abrir um arquivo com File.Open:
  - Se o usuário informar um caminho de arquivo inexistente, será lançada uma exceção do tipo FileNotFoundException;
  - Se o arquivo informado já estiver sendo usado por outro programa, será lançada uma exceção do tipo IOException;
  - Se o usuário teclar Enter (string vazia) ou incluir caracteres especiais como <, >, ?, *, será lançada uma exceção do tipo ArgumentException.
- O bloco catch, escrito dessa forma, irá capturar todo tipo de exceção.

```
Console.Write("Informe o caminho do arquivo de texto que deseja abrir: ");
var caminhoArquivo = Console.ReadLine() ?? "";
try
{
    var streamArquivo = File.OpenText(caminhoArquivo);
    Console.WriteLine("Arquivo aberto com sucesso!");
}
catch (Exception e)
{
    Console.WriteLine($"Falha ao abrir arquivo: {e.Message}, {e.GetType()}");
}
```

Tratamento de Exceções – Exemplo 2

Neste exemplo os erros são tratados separadamente, com base no tipo específico da exceção lançada.

```
Console.Write("Informe o caminho do arquivo de texto que deseja abrir: ");
var caminhoArquivo = Console.ReadLine() ?? "";
try
{
    var meuArquivo = File.OpenText(caminhoArquivo);
    Console.WriteLine("Arquivo aberto com sucesso!");
}
catch (FileNotFoundException)
{
    Console.WriteLine("Arquivo não encontrado!");
}
catch (IOException e)
{
    Console.WriteLine($"O arquivo não pode ser aberto: {e.Message}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Nome de arquivo inválido: {e.Message}");
}
```

Tratamento de Exceções – Exemplo 3

```
Console.Write("Informe o caminho do arquivo que deseja abrir: ");
var caminhoArquivo = Console.ReadLine() ?? "";
StreamReader? streamReader = null; // declara variável
try
{
    // Abre arquivo existente para leitura de texto.
    // Um 'stream' representa um canal de entrada e saída por onde os dados (bytes)
    // são gravados ou lidos sequencialmente.
    streamReader = File.OpenText(caminhoArquivo);
    Console.WriteLine("Arquivo aberto com sucesso!");

    // Lê uma linha de texto do arquivo. Esta operação também pode lançar exceções.
    var primLinhaTexto = streamReader.ReadLine();
    Console.WriteLine($"Texto lido com sucesso: {primLinhaTexto}");
}
catch (Exception e) // Captura todo tipo de exceção (tipo genérico Exception)
{
    // Mostra a mensagem de erro associada à exceção e o tipo da exceção
    Console.WriteLine($"Falha ao abrir ou ler do arquivo: {e.Message}, {e.GetType()}");
}
finally
{
    // Fecha o arquivo aberto, ocorrendo falha ou não na gravaçao do texto.
    if (streamReader != null) streamReader.Close();
}
```

Lançando uma Exceção com throw

- Nos exemplos apresentados anteriormente as exceções são lançadas pelos próprios métodos das classes do framework .NET (File.Open, stream.WriteLine etc.).
- Entretanto, há situações onde é necessário lançar exceções manualmente, a partir do próprio código.
- Para esses casos pode-se utilizar a palavra reservada throw

Lançando uma Exceção com throw – Exemplo

```
class ValidacaoDados
{
    public static void ValidaDadosCliente(string nome, int idade, string estadoCivil)
    {
        if (nome is null)
            throw new ArgumentNullException("O nome do cliente não pode ser nulo.");
        if (nome.Length < 3)
            throw new ArgumentException("O nome precisa ter pelo menos 3 caracteres.");
        if (idade < 0 || idade > 150)
            throw new ArgumentOutOfRangeException("A idade deve estar entre 0 e 150 anos.");

        string[] estadosPermitidos = [ "solteiro", "casado", "divorciado", "viuvo" ];
        if (!estadosPermitidos.Contains(estadoCivil))
            throw new ArgumentException("Estado civil inválido.");
    }
}
```

```
var nome = "Fulano"; var idade = 200; var estadoCivil = "solteiro";
try {
    ValidacaoDados.ValidaDadosCliente(nome, idade, estadoCivil);
    BancoDeDados.SalvaDadosCliente(nome, idade, estadoCivil);
}
catch (Exception e) {
    Console.WriteLine($"Falha na operação: {e.Message}");
}
```

Referências

- Joseph Albahari. **C# 12 in a Nutshell**: The Definitive Reference. 1ª ed., 2023.
- Mark J. Price. **C# 9 and .NET 5 – Modern Cross-Platform Development**: Build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code, 5ª ed., 2020.
- learn.microsoft.com/en-us/dotnet/csharp/fundamentals