

PROGRAMAÇÃO 3

SEMANA #2



Programação 3 - APR-211

Masterclass Semana 2

Programação Orientada a Objetos e Estruturas de Dados

Prof. José Paulo R. de Lima



Agenda

- Revisão semana 1
- Programação Orientada a Objetos com C#
- Genéricos
- Comparações com Java
- Estruturas de dados comuns (Listas, Arrays e Dicionários)

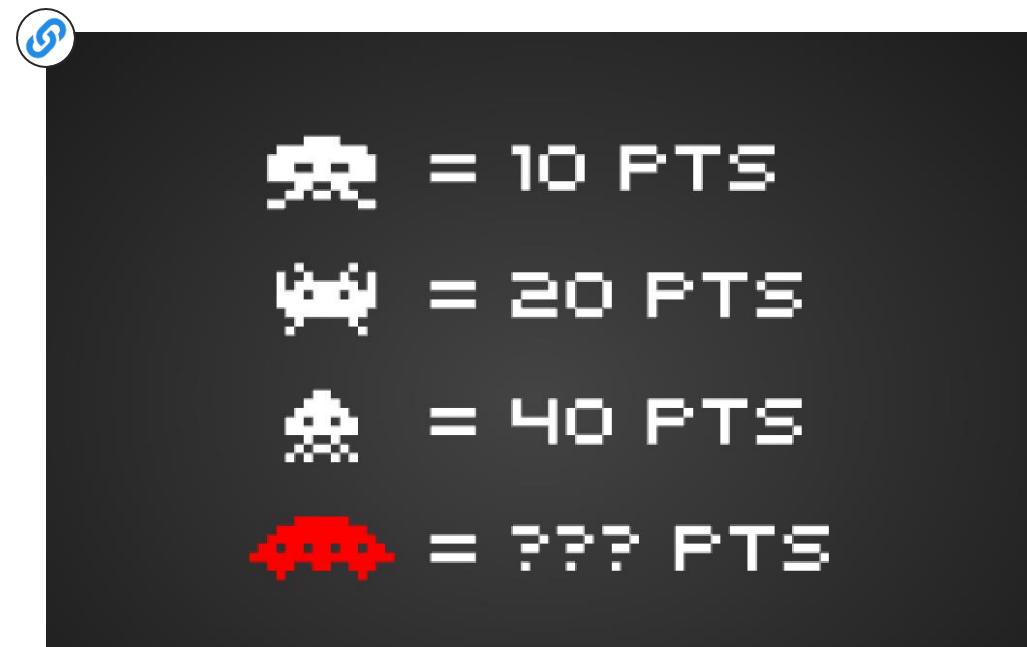
Projeto de Curso: “Space Invaders”



Space Invaders

- Nosso Projeto de fim de curso é uma recriação do jogo “Space Invaders”. Usaremos um aplicativo de desktop para conseguir isso.
- O objetivo é demonstrar o uso de componentes visuais e interagir com eles por meio de eventos. Gerenciar processos (movimentações) sem afetar o uso do aplicativo, além do uso de arquivos como mecanismos para salvar e carregar informações.
- O jogo permitirá que você controle uma nave que contém um laser, que pode atirar contra as ondas de naves alienígenas dessa forma para aumentar a pontuação. Para aumentar a pontuação, leve em conta os seguintes elementos e seus pesos:

[Aqui](#) você pode encontrar uma versão online do jogo



Space Invaders

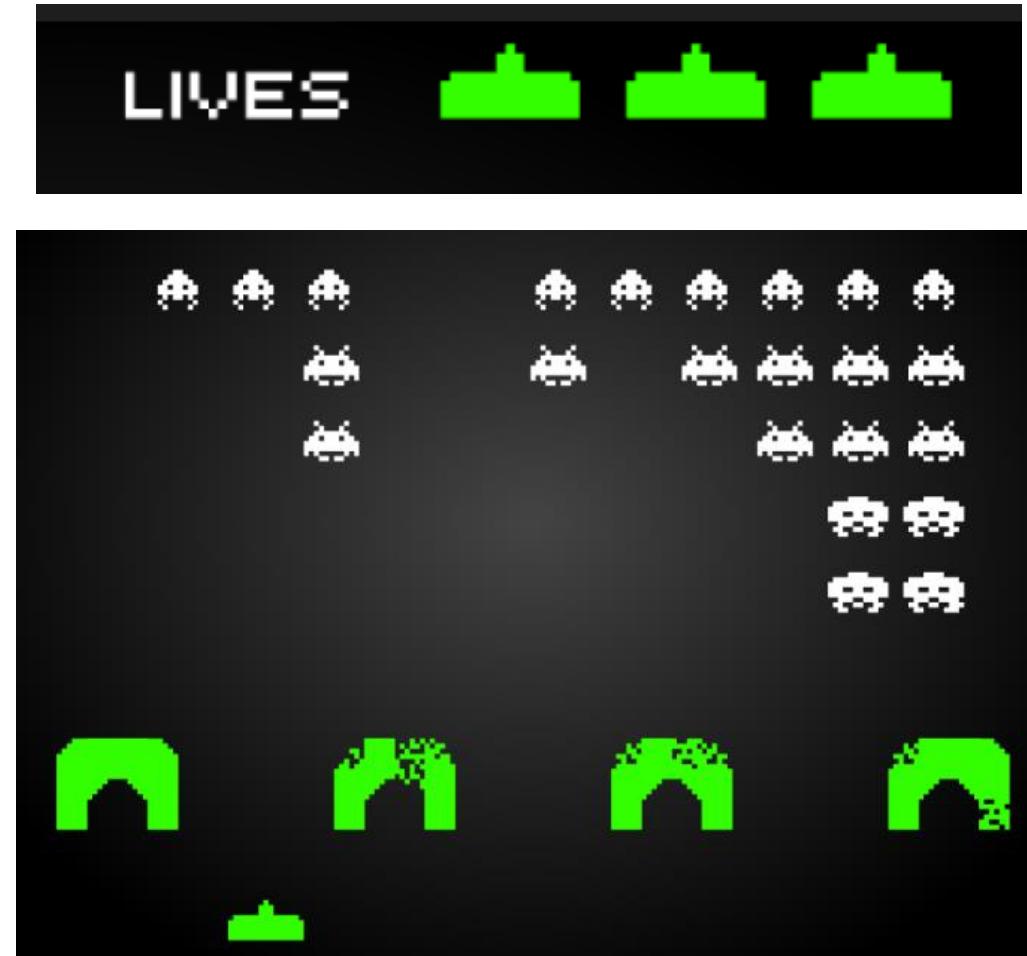
- As naves alienígenas vermelhas, se movem aleatoriamente começando da esquerda para a direita ou vice-versa e saindo do quadro, aparecem de vez em quando, 1 vez por 2 minutos aproximadamente, os pontos que eles lhe dão ao atirar variam.

Quando o jogo termina

- O jogo terminará quando as vidas do jogador acabarem
- Ou terminará quando as naves alienígenas alcançarem o jogador.

Outros elementos importantes

- Serão 4 blocos de proteção que ajudarão o jogador com os tiros dos alienígenas. Estes mudarão sua cor de branco para chumbo, à medida que mais dano eles recebem dos tiros alienígenas, eventualmente desaparecendo



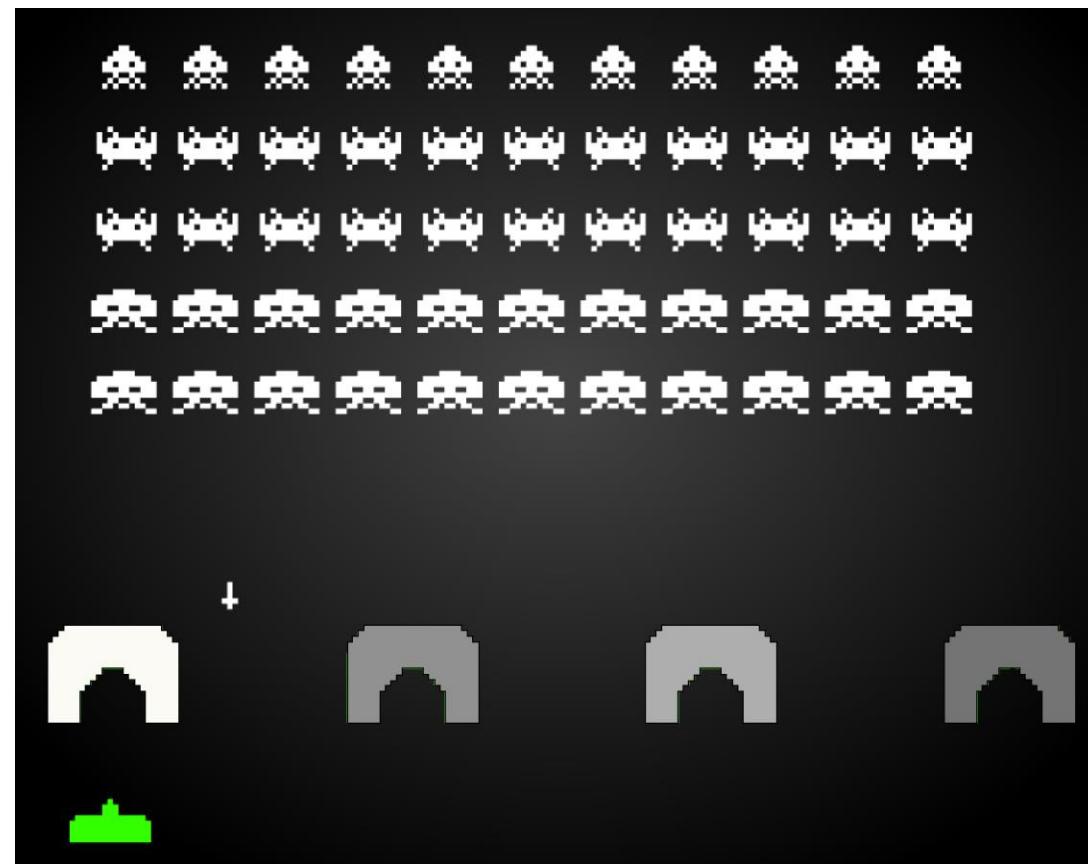
Space Invaders

Movimento do Jogador

- O jogador será capaz de se mover da esquerda para a direita pressionando as setas para a esquerda e direita, respectivamente
- O botão Espaço (Space) será usado para atacar.

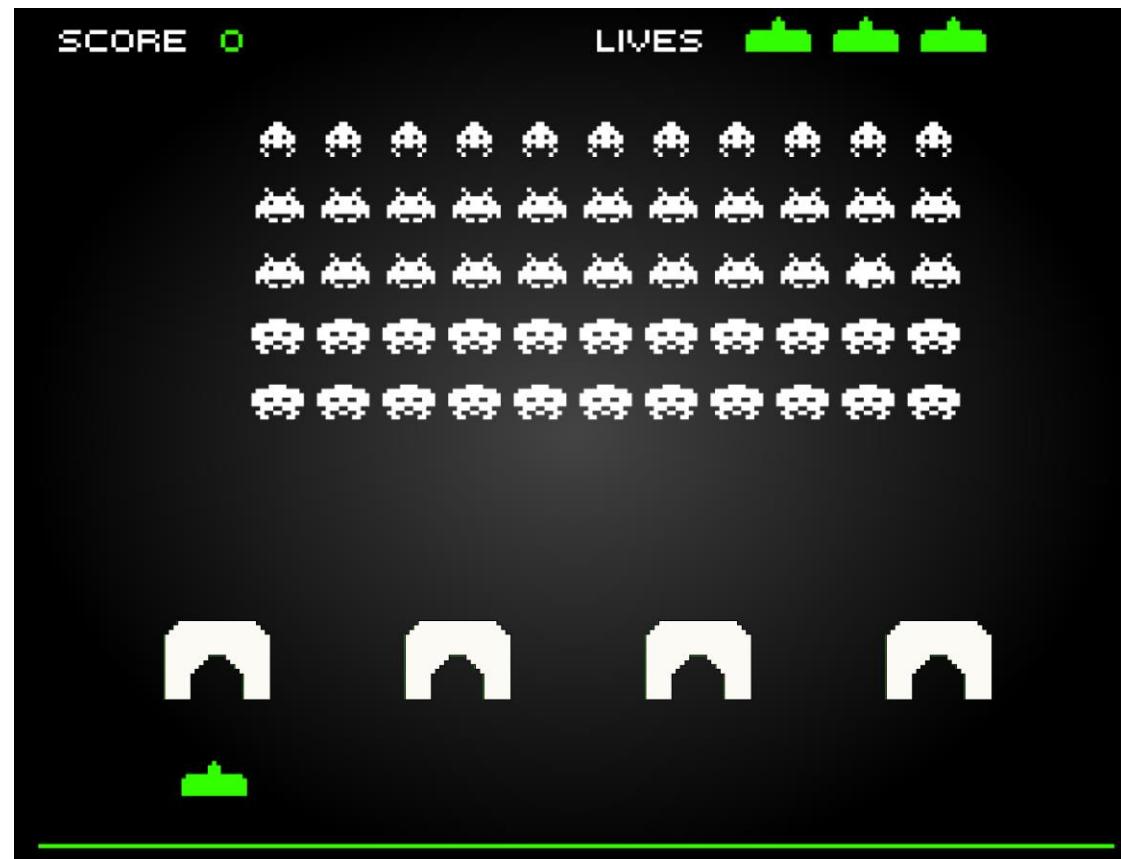
Movimento das Naves Alienígenas

- Ela se move da esquerda para a direita, uma vez que atinge a borda, em seguida, ela se move para baixo uma posição e se move para a borda oposta.
- Cada vez que há um rolo para baixo, a velocidade do movimento aumenta um pouco, assim como a velocidade do fogo deles.
- A única nave que ataca é o que vale 40 PTS.



Space Invaders

- Os blocos de proteção se desgastam toda vez que um tiro é atingido, seja de uma nave alienígena ou do jogador.
- O jogador pode disparar novamente quando seu tiro anterior acertar ou de outra forma quando exceder o limite superior do quadro do jogo.
- Alienígenas não podem colidir com os blocos de proteção.
- Naves alienígenas são destruídas com 1 tiro.
- Quando todas as naves alienígenas são destruídas, uma nova onda é gerada movendo a posição para baixo para os primeiros 5 e aumentando ligeiramente a velocidade a partir da quinta em diante.



Space Invaders

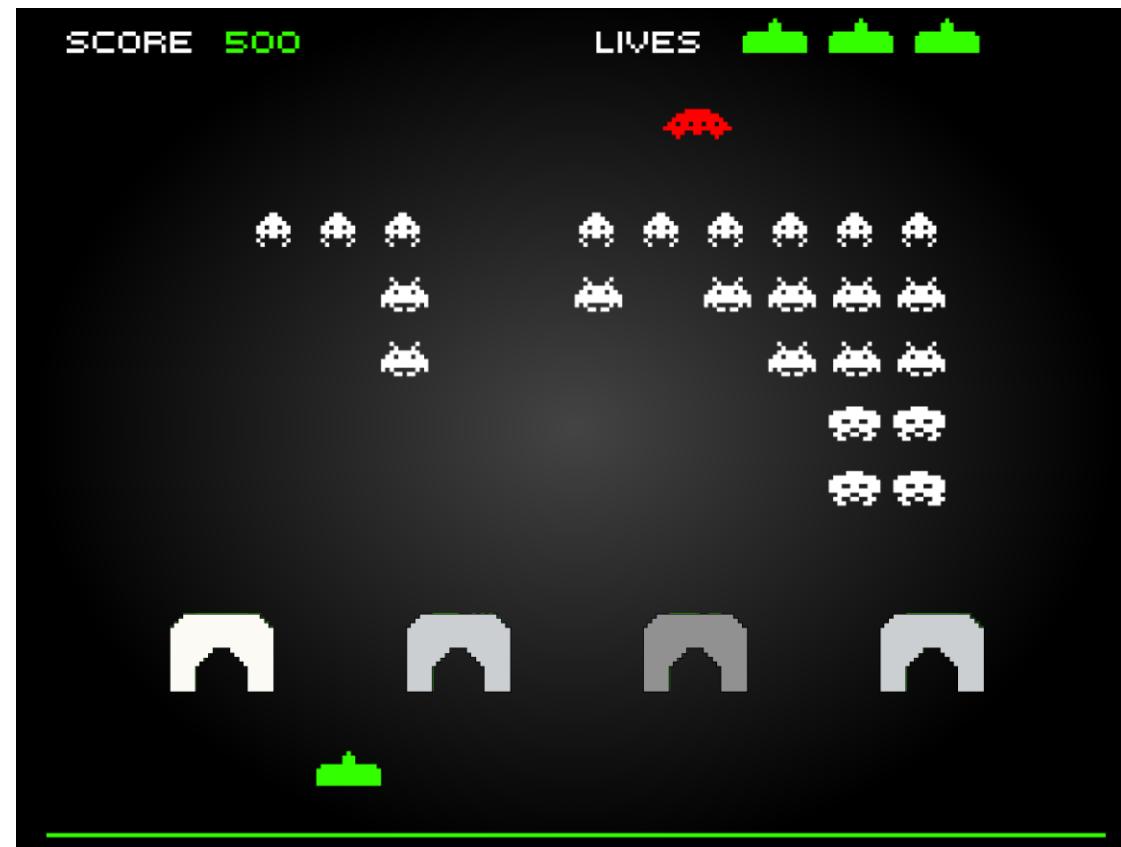
- A cada 1000 pontos, o número de vidas é aumentado em uma, até um máximo de 6.

Sobre a pontuação

- A pontuação é exibida no canto superior esquerdo
- Ela é incrementada seguindo a tabela mostrada acima cada vez que um alienígena é destruído.

O que acontece após o fim do jogo

- O jogador tem a opção de salvar sua pontuação adicionando um apelido.
- O jogador tem a opção de voltar a jogar.
- Se o jogador não quiser jogar novamente, a página inicial será exibida.



Space Invaders

Tela inicial

- A tela inicial dará as seguintes opções::
- Iniciar um novo jogo.
- Ver a tabela com os placares.
- Ver os controles do jogo.

Cada ação no jogo deve ser acompanhada por um som representativo.

As informações do painel de pontuação devem ser salvas em um arquivo de texto.



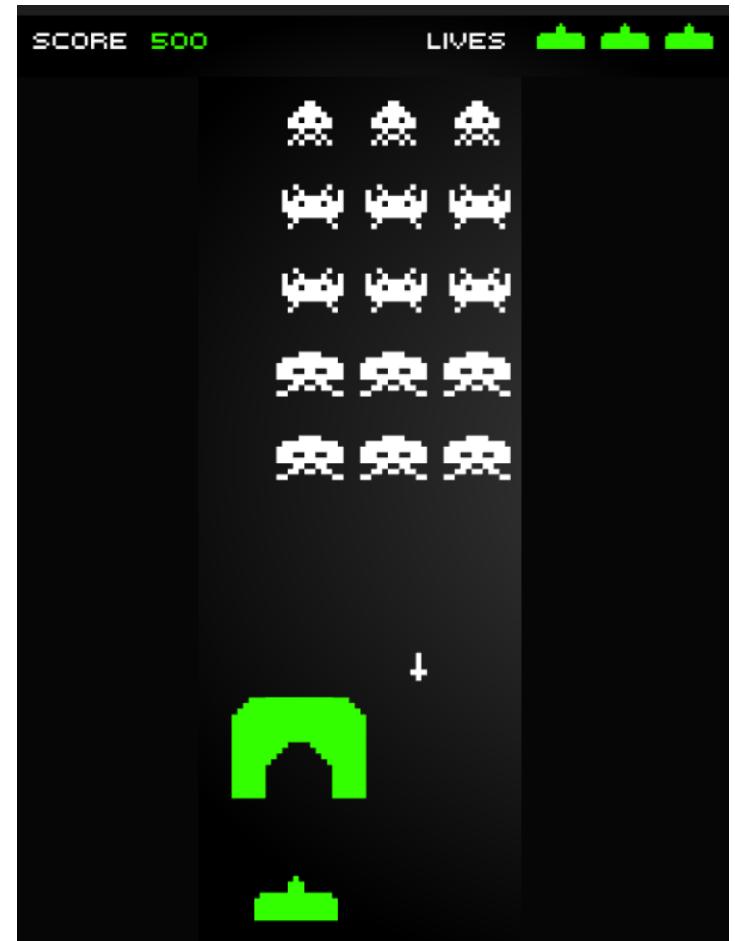
Projeto Midterm: “Space Invaders” first stage



Space Invaders first stage

O projeto a ser apresentado na semana 4 do módulo consiste em uma versão inicial de "Space Invaders". Apenas o seguinte deve ser considerado:

- Não há necessidade de naves alienígenas se moverem.
- Basta contar com as naves mostradas e o bloco de proteção mostrado
- O jogador deve ser capaz de se mover e atirar
- A pontuação será atualizada toda vez que um navio for destruído. A pontuação deve ser salva na memória quando o jogo terminar (isso significa que quando o aplicativo fechar as pontuações serão perdidas)
- O bloco de proteção deve poder ser destruído pelo usuário
- Cada ação possível deve emitir um som correspondente.
- Deve haver uma tela inicial com os itens já descritos.
- O jogo termina quando você atinge 500 pontos. (Pode haver ondas)





Revisão: Semana 1

Ecossistema .NET



O .NET é uma plataforma de desenvolvedor gratuita, multiplataforma e de código aberto projetada para criar muitos tipos diferentes de aplicativos.

O .NET foi criado em um ambiente de tempo de execução de alto desempenho que muitos aplicativos de grande escala usam na produção.

O ecossistema .NET fornece soluções em diferentes campos, por exemplo:

Desktop

Permite criar aplicações nativas para Windows e MacOS ou desenvolvimento de app que podem ser executados em qualquer lugar com tecnologia web.

Web

Permite criar aplicações e serviços web para Windows, Linux, MacOS e Docker.

Cloud

Permite consumir serviços na nuvem ou criar e implementar seus próprios serviços de nuvem.

Games

Permite desenvolver jogos 2D e 3D para os computadores, smartphones e consoles mais populares.

Mobile

Permite o uso de código base para construir aplicações nativas para iOS e Android.

AI and IoT

Permite criar aplicações IoT com suporte nativo para Raspberry Pi e outros microcontroladores. Também permite agregar algoritmos de processamento e modelos preditivos em suas aplicações.



Ecossistema .NET

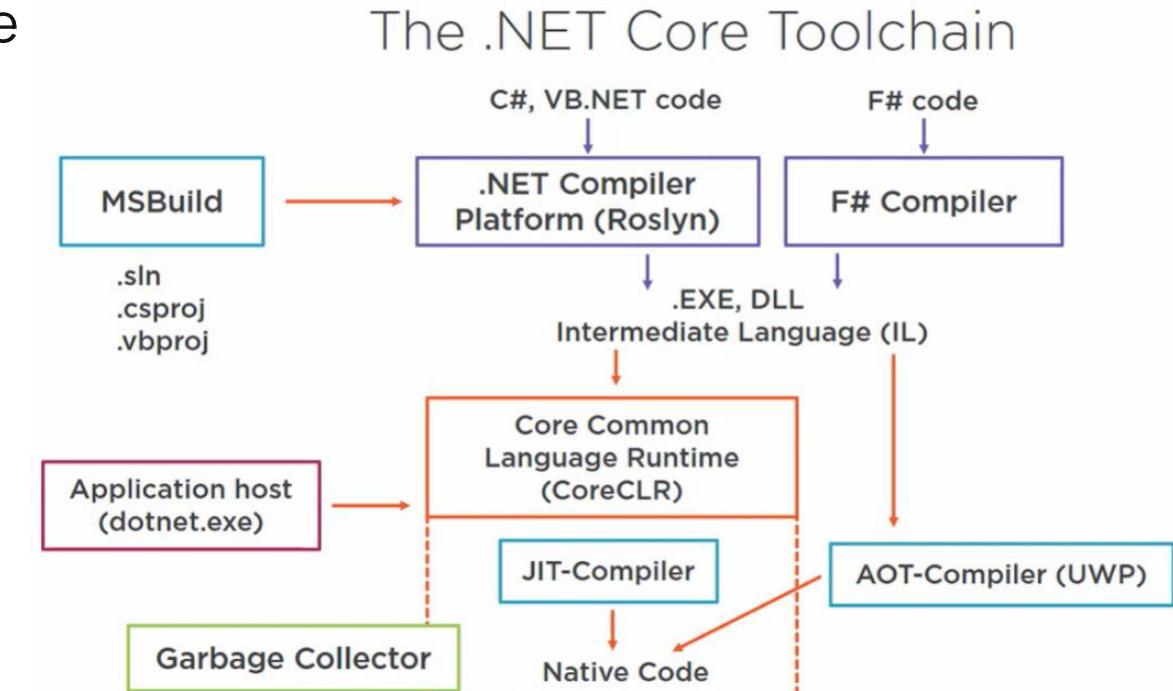
Processos de Compilação e Execução

Compilação

- O compilador C#: Traduzir o código-fonte do C# para a CIL.
- Produz arquivos .dll e .exe.
- Ocorre a compilação Just-in-time (JIT): Tradução de CIL para código de máquina.

Execução

- Usando compilação JIT intercalada.
- Em mono: ativação explícita do interpretador.
- No Windows: ativação do interpretador transparente.



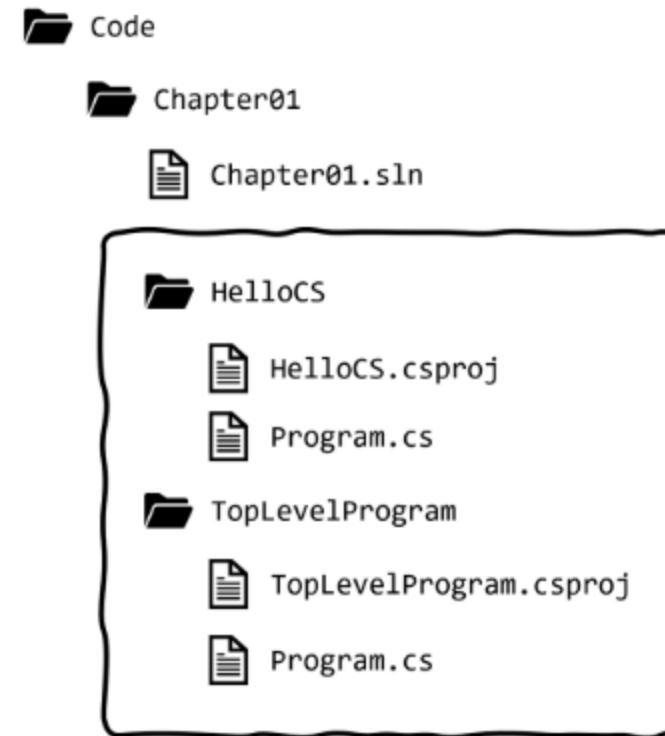
Fundamentos de C#



Organizando uma aplicação em C#

Uma aplicação em C# é organizada da seguinte maneira:

- Uma solução contém um ou mais projetos.
- Cada projeto contém um ou mais *namespaces*.
- O código-fonte de um aplicativo é colocado nos *namespaces* dos projetos.





Fundamentos de C#

Características dos Tipos Numéricos



C# type/keyword	Range	Size	.NET type	
<code>sbyte</code>	-128 to 127	Signed 8-bit integer	<code>System.SByte</code>	
<code>byte</code>	0 to 255	Unsigned 8-bit integer	<code>System.Byte</code>	
<code>short</code>	-32,768 to 32,767	Signed 16-bit integer	<code>System.Int16</code>	
<code>ushort</code>	0 to 65,535	Unsigned 16-bit integer	<code>System.UInt16</code>	
<code>int</code>	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	<code>System.Int32</code>	
<code>uint</code>	0 to 4,294,967,295	Unsigned 32-bit integer	<code>System.UInt32</code>	
<code>long</code>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	<code>System.Int64</code>	
<code>ulong</code>	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	<code>System.UInt64</code>	
<code>nint</code>	Depends on platform (computed at runtime)	Signed 32-bit or 64-bit integer	<code>System.IntPtr</code>	
<code>nuint</code>	Depends on platform (computed at runtime)	Unsigned 32-bit or 64-bit integer	<code>System.UIntPtr</code>	
<code>float</code>	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes	<code>System.Single</code>
<code>double</code>	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes	<code>System.Double</code>
<code>decimal</code>	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes	<code>System.Decimal</code>

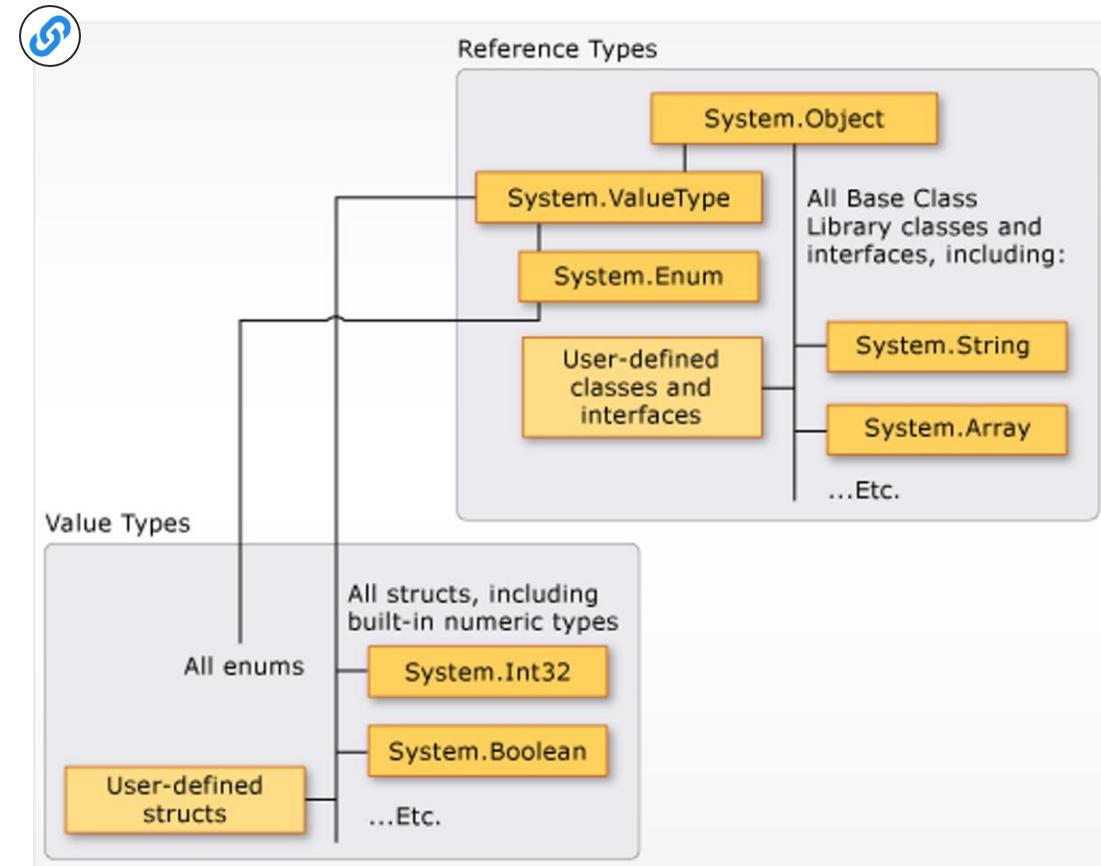
Fundamentos de C#



Sistema de tipos

O Common Type System (CTS) é um conjunto de regras e definições que descrevem os tipos de dados que podem ser usados em C#. O CTS define tipos de dados primitivos, estruturas, classes, enumerações, tipos de referência e tipos de valor.

A seguir exploraremos mais sobre a classificação dos tipos por valor e por referência.



Fundamentos de C#

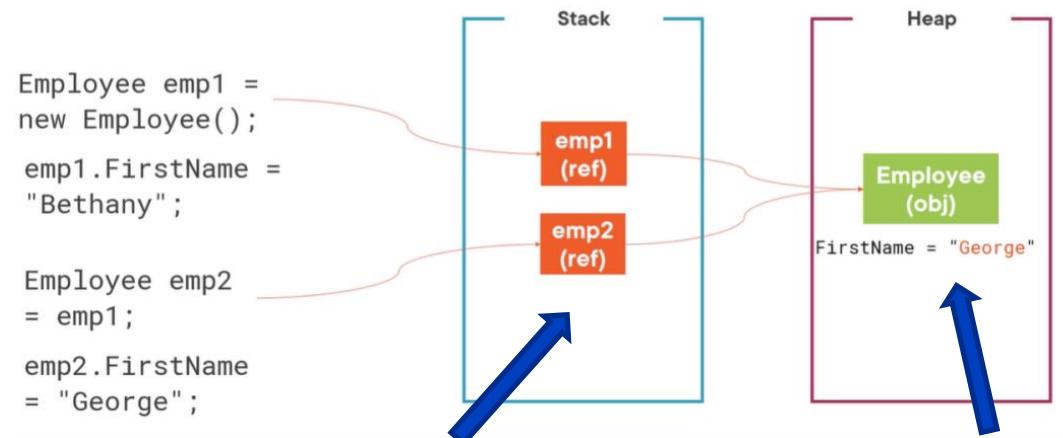
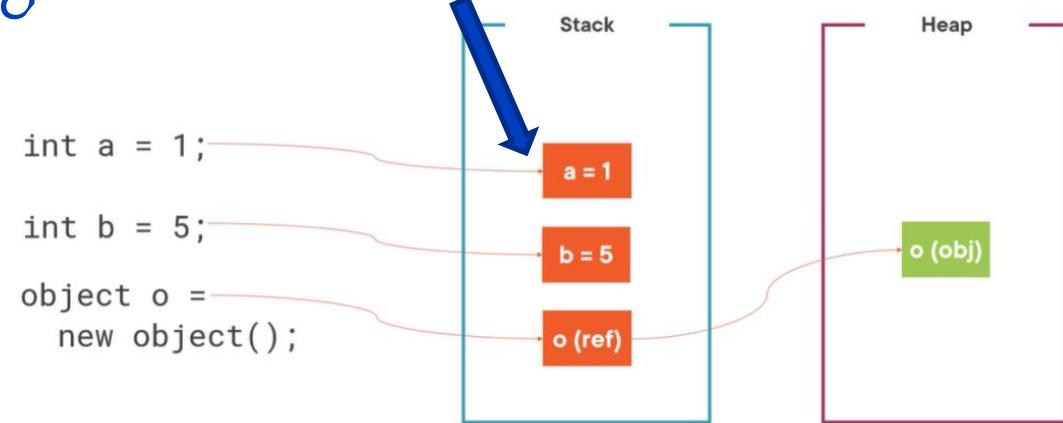


Tipos por valor e por referência

Os tipos de valor são derivados de `System.ValueType` e os dados são armazenados diretamente na pilha (*Stack*). Quando você atribui uma variável de valor de tipo a outra variável, o valor da primeira variável é copiado para a segunda.

Ao declarar uma variável de um *reference type*, ela contém o valor *null* até que um valor seja atribuído (por exemplo, usando o operador *new*). Quando você atribui uma variável de tipo de referência a outra variável, a referência da primeira variável é copiada para a segunda.

Os tipos por valor são armazenados na *stack*



Tipos por *ref* armazenam uma referência na *stack*

O valor é armazenado no *heap*

Fundamentos de C#



Resumo de Operadores

Em geral, a maioria dos operadores como os aritméticos, relacionais, lógicos de incremento e decremento, comparação são os mesmos para C# e Java.

Quanto ao seu uso com tipos de referência, o C# permite que você use esses operadores sem qualquer problema.

Level	Category	Operators	Associativity (binary/tertiary)
14	Primary	<code>x.y f(x) a[x] x++ x-- new typeof checked unchecked default delegate</code>	<i>left to right</i>
13	Unary	<code>+ - ! ~ ++x --x (T)x true false sizeof</code>	<i>left to right</i>
12	Multiplicative	<code>* / %</code>	<i>left to right</i>
11	Additive	<code>+</code>	<i>left to right</i>
10	Shift	<code><< >></code>	<i>left to right</i>
9	Relational and Type testing	<code>< <= > >= is as</code>	<i>left to right</i>
8	Equality	<code>== !=</code>	<i>left to right</i>
7	Logical/bitwise and	<code>&</code>	<i>left to right</i>
6	Logical/bitwise xor	<code>^</code>	<i>left to right</i>
5	Logical/bitwise or	<code> </code>	<i>left to right</i>
4	Conditional and	<code>&&</code>	<i>left to right</i>
3	Conditional or	<code> </code>	<i>left to right</i>
2	Null coalescing	<code>??</code>	<i>left to right</i>
1	Conditional	<code>? :</code>	<i>right to left</i>
0	Assignment or Lambda expression	<code>= *= /= %= += -= = <<= >>= &= ^= = =></code>	<i>right to left</i>

Fundamentos de C#



Conversões

Conversão implícita

```
int a = 10;
double b = a; // an int can be safely cast into a
               double
WriteLine(b);      O resultado é 10
```

Conversão explícita

```
double c = 9.8;
int d = c; // compiler gives an error for this line
           O compilador dá um erro
WriteLine(d);
```

```
int d = (int)c;
WriteLine(d); // d is 9 losing the .8 part
```

```
using static System.Console;
```

Importando o
namespace

Usando o tipo System.Convert

```
using static System.Convert;
double g = 9.8;
int h =ToInt32(g); // a method of System.Convert
WriteLine($"g is {g} and h is {h}");
```

g is 9.8 and h is 10

← Resultado

Usando el operador cast
Resultado es 9

Fundamentos de C#

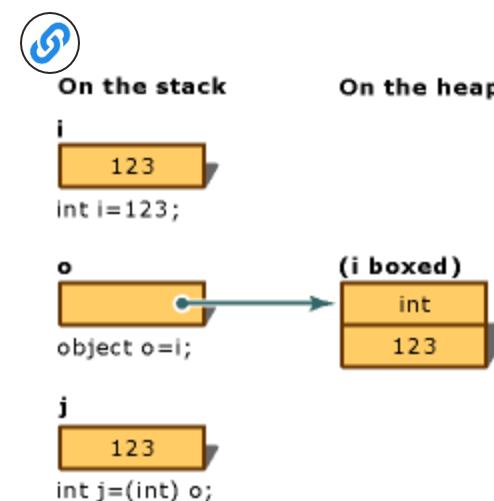
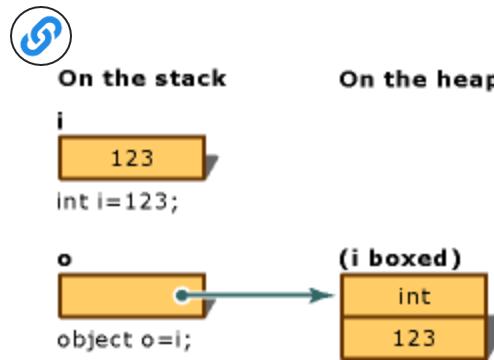


Boxing e unboxing

A conversão *boxing* é uma conversão implícita de um tipo por valor no tipo de *object* ou em qualquer tipo de interface implementada por esse tipo de valor. Quando aplicada, uma instância de objeto é atribuída ao *heap* e o valor é copiado para o novo objeto.

A conversão de *unboxing* é uma conversão explícita do tipo de objeto em um tipo por valor.

Primeiro, ele verifica se o tipo do valor de *unboxing* é do tipo salvo no *heap* e, em seguida, copia o valor da instância para a variável de tipo por valor.



```
using System;  
  
public class BoxingTest{  
    public static void Main(){  
        int i = 15, j, k;  
  
        bool b = false, c, d;  
        Object obj1 = i, // boxing of the value of i  
              obj2 = b; // boxing of the value of b  
  
        j = (int) obj1; // unboxing obj1  
        c = (bool) obj2; // unboxing obj2  
  
        // k = i + obj1; // Compilation error  
        // d = b && obj2; // Compilation error  
  
        k = i + (int) obj1;  
        d = b && (bool) obj2;  
  
        Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}",  
                          i, obj1, b, obj2, j, c, k, d);  
    }  
}
```

Boxing

Unboxing



Fundamentos de C#

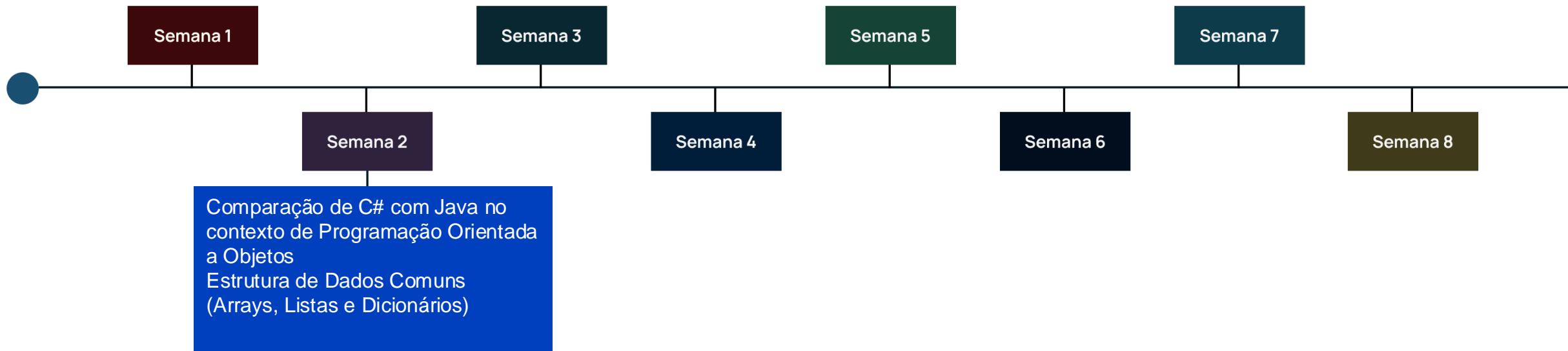
Comparação entre C# e JAVA

Feature	C#	Java
Plataforma	.NET	Java Virtual Machine (JVM)
Runtime	Common Language Runtime (CLR)	Java Runtime Environment (JRE)
Geração de código IL	O compilador gera MSIL (Microsoft Intermediate Language)	O compilador gera bytecode Java
Object-oriented	Sim	Sim
Functional	Sim, C# é uma linguagem multiparadigma	Não
Strongly typed	Sim	Sim
Go-to statement	É suportado	Não é suportado
Tools	Visual Studio, Visual Studio Code, Rider	Eclipse, IntelliJ IDEA, Visual Studio Code

Semana 2



Programação 3

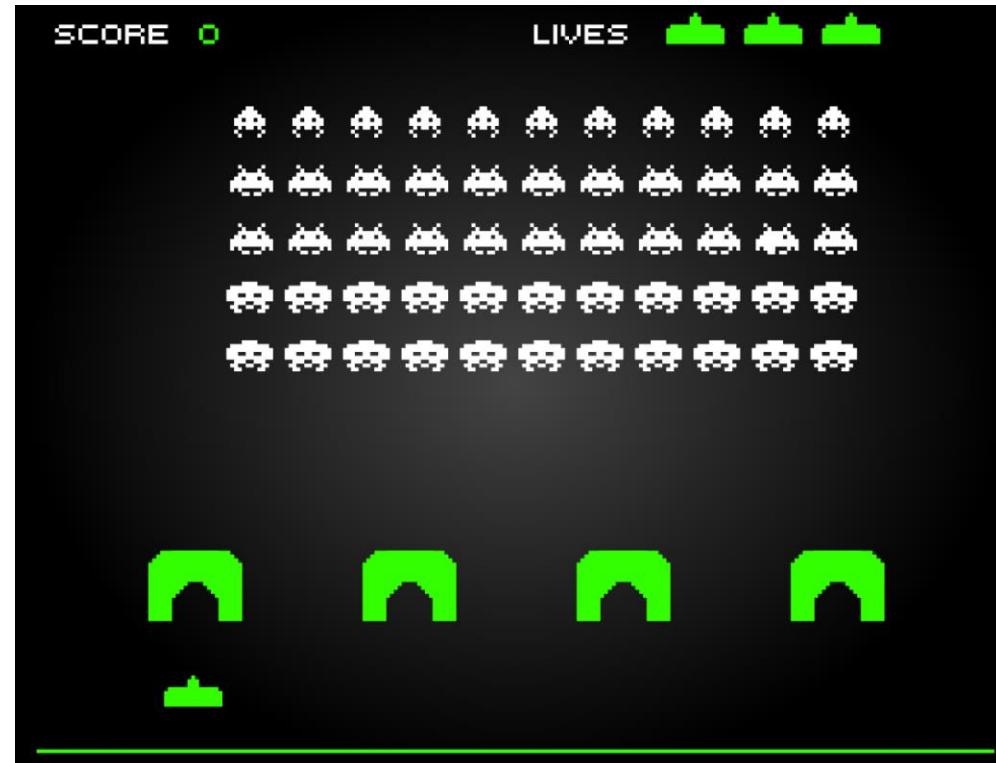


Clases en directo

Projeto Capstone – Space Invaders



- Lembre-se de progredir a cada semana neste projeto e revisá-lo com seu tutor (*practitioner*). O progresso semanal será levado em conta em sua pontuação com o tutor.
- Após este curso, considere os seguintes aspectos:
 - Eu poderia usar OOP e estruturas de dados para executar a lógica do Score?



Classes e structs

Programação Orientada a Objetos



Criação e instanciação de classes

```
namespace Packt.Shared
{
    public class BankAccount
    {
        public string AccountName; // instance member
        public decimal Balance; // instance member
        public static decimal InterestRate; // shared
        member
    }
}

var account = new BankAccount();
var newBankAccount = new BankAccount
{
    AccountName = "Personal_2025",
    Balance = 100000M,
};

account.AccountName = "Company_2020";
Console.WriteLine(newBankAccount.AccountName);
```

Declaração de namespace

Declaração de classe

Atributo da classe

Atributo estática da classe

Instanciando uma classe, usando o construtor padrão

Instanciando uma classe atribuindo valores aos atributos.

Reatribuindo o valor de um atributo da instância

Usando um atributo de instância de classe

Programação Orientada a Objetos



Membros de uma classe

- Campos (Fields), estão associados à classe. Existem as seguintes categorias:
 - Constantes
 - ReadOnly
- Construtores, métodos necessários que são executados ao inicializar uma classe

```
public partial class BankAccount
{
    private string _accountNumber;
    private decimal _balance;
    private List<Transaction> _transactions;
}
```

```
public partial class BankAccount
{
    public BankAccount(string accountNumber)
    {
        _accountNumber = accountNumber;
        _balance = 0.0m;
        _transactions = new List<Transaction>();
    }
}
```

Programação Orientada a Objetos



Membros de uma classe

- Propriedades, atributos que permitem a escrita e leitura de recursos de classe

The screenshot shows a code editor window with the title "Properties". The code defines a partial class "BankAccount" with two properties: "AccountNumber" and "Balance". The "AccountNumber" property is annotated with a note "Allows reading and writing". It has a get accessor that returns the value of the private field "_accountNumber" and a set accessor that updates the value of "_accountNumber" to the provided "value". The "Balance" property has a get accessor that returns the value of the private field "_balance" and a private set accessor that updates the value of "_balance" to the provided "value".

```
public partial class BankAccount
{
    // Allows reading and writing
    public string AccountNumber
    {
        get { return _accountNumber; }
        set { _accountNumber = value; }
    }

    public decimal Balance
    {
        get { return _balance; }
        private set { _balance = value; }
    }
}
```

- Métodos, ações e comportamentos de uma classe

The screenshot shows a code editor window with the title "Methods". The code defines a partial class "BankAccount" with a single method "Deposit". The "Deposit" method takes a "decimal amount" as a parameter. It first checks if the "amount" is less than or equal to zero, throwing an "ArgumentException" if it is. Then, it adds the "amount" to the current balance and adds a new transaction record to the list of transactions.

```
public partial class BankAccount
{
    // Adds the specified amount to the account balance
    public void Deposit(decimal amount)
    {
        if (amount <= 0)
            throw new ArgumentException("Amount must be positive.");

        _balance += amount;
        _transactions.Add(new Transaction(amount, "Deposit"));
    }
}
```

Programação Orientada a Objetos



Membros de uma classe

- Operadores, conversões e expressões de operador suportados pela classe, seu uso é incomum
- Finalizadores, ações executadas antes que a instância de classe seja descartada.

Operators

```
public partial class BankAccount
{
    // Defines an implicit conversion operator from BankAccount
    // to decimal to get the account balance directly.
    public static implicit operator decimal(BankAccount account)
    {
        return account.Balance;
    }
}
```

Finalizer

```
public partial class BankAccount
{
    ~BankAccount()
    {
        // Cleanup code here
    }
}
```

Você também pode ver o uso de classes parciais(partial), Uma classe parcial permite que a declaração de membro seja separada em vários arquivos e, em tempo de compilação, seja montada para formar a classe completa. Eles são úteis para ferramentas que geram código automaticamente, permitindo que você o estenda conforme necessário.

Programação Orientada a Objetos



Membros de uma classe

- Indexadores, ações associadas ao processo de indexação de instâncias de classe.
- Tipos, são tipos de dados aninhados

Indexers

```
public partial class BankAccount
{
    // Allows access to the balance
    // using the square bracket notation.
    public decimal this[int index]
    {
        get => _balance;
        set => _balance = value;
    }
}
```

Types

```
public partial class BankAccount
{
    public class Transaction
    {
        public decimal Amount { get; }
        public DateTime Date { get; }
        public string Type { get; }

        public Transaction(decimal amount, string type)
        {
            Amount = amount;
            Date = DateTime.Now;
            Type = type;
        }
    }
}
```

Programação Orientada a Objetos



Membros de uma classe

- Eventos, uma notificação que é enviada a um objeto quando algo acontece.

```
public partial class BankAccount
{
    public event EventHandler BalanceChanged;

    // It Is triggered whenever the balance of the account changes.
    protected virtual void OnBalanceChanged()
    {
        BalanceChanged?.Invoke(this, EventArgs.Empty);
    }
}
```

Cobriremos Eventos
extensivamente, em uma aula
futura

Programação Orientada a Objetos



Constants

```
// constants  
public const string Species = "Homo Sapiens";
```

Declaração de Constantes

```
WriteLine($"{bob.Name} is a {Person.Species}");
```

O uso de constantes é semelhante a um membro estático

Uso de readonly

```
Person blankPerson = new();  
WriteLine(format:  
    "{0} of {1} was created at {2:hh:mm:ss} on a  
    {3:dddd}.",  
    arg0: blankPerson.Name,   
    arg1: blankPerson.HomePlanet,  
    arg2: blankPerson.Instantiated);
```

Resultado

Unknown of Earth was created at 11:58:12 on a Sunday

```
// read-only fields  
public readonly string HomePlanet = "Earth";   
public readonly DateTime Instantiated;  
// constructors  
public Person()  
{  
    // set default values for fields  
    // including read-only fields  
    Name = "Unknown";  
    Instantiated = DateTime.Now;  
}
```

Resultado

Bob Smith is a Homo Sapiens

Declaração de readonly

Atribuição de valores,
somente no construtor

Programação Orientada a Objetos



Declarando propriedades de diferentes formas

```
private string favoritePrimaryColor;
public string FavoritePrimaryColor
{
    get
    {
        return favoritePrimaryColor;
    }
    set
    {
        switch (value.ToLower())
        {
            case "red":
            case "green":
            case "blue":
                favoritePrimaryColor = value;
                break;
            default:
                throw new System.ArgumentException(
                    $"{value} is not a primary color. " +
                    "Choose from: red, green, blue.");
        }
    }
}
```

Campo privado,
armazena o estado da
propriedade

Método Get, leitura do
valor da propriedade

Método Set, escrita
do valor da
propriedade

```
// a property defined using C# 1 - 5 syntax
public string Origin
{
    get
    {
        return $"{Name} was born on {HomePlanet}";
    }
}

// two properties defined using C# 6+ lambda
// expression body syntax
public string Greeting => $"{Name} says 'Hello!'";
public int Age => System.DateTime.Today.Year -
DateOfBirth.Year;
```

```
public string FavoriteIceCream { get; set; } // auto-
syntax
```

Propriedade somente
leitura, Getter

Propriedade
usando
expressões
com
operadores
lambda =>

Propriedades
com sintaxe
por padrão

Programação Orientada a Objetos



Structs

Um tipo [struct](#), é um tipo de dados por valor que pode encapsular propriedades e funções relacionadas.

```
public struct Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public void Display()
    {
        Console.WriteLine($"Point: ({X}, {Y})");
    }
}
```

Declaracao da Struct

Propriedades

Construtor, geralmente deve inicializar as propriedades

Método

```
public class Program
{
    public static void Main()
    {
        Point p1 = new Point(10, 20);
        Point p2 = p1; // p2 gets a copy of p1's values

        p1.Display(); // Output: Point: (10, 20)
        p2.Display(); // Output: Point: (10, 20)

        p1.X = 30;

        p1.Display(); // Output: Point: (30, 20)
        p2.Display(); // Output: Point: (10, 20)
    }
}
```

Instanciação de struct

Invocação do método

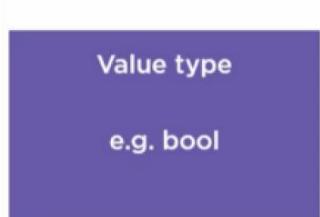
Usando propriedades

Programação Orientada a Objetos

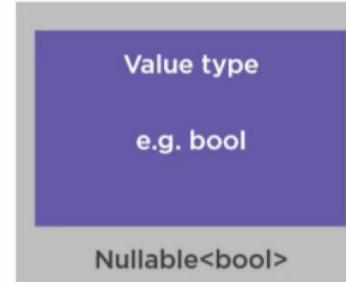


Tipos nullable

Instâncias do System.Nullable struct. Um cara que aceita valores NULL. Você pode representar o intervalo correto de valores para seu tipo de valor subjacente, além de um valor nulo adicional.



True
False



True
False
null

Inicialização de tipo nullable

Nullable<int> i = null;

Para versões mais antigas do C#

Or:

int? i = null;

Para versões modernas do C#

Or:

var i = (int?)null;

Para versões modernas do C#, a inferência de tipo é usada implementando a palavra var

For non-null values:

Nullable<int> i = 0;

Para versões mais antigas do C#

Or:

int? i = 0;

Para versões modernas do C#

Programação Orientada a Objetos



Tipos nullable

Verifique se um tipo Nullable tem valores

```
int? i = null;  
  
if (i != null)           ← Usando operador !=  
{  
    Console.WriteLine("i is not null");  
}  
else  
{  
    Console.WriteLine("i is null");  
}
```

Which is the same as:

```
if (i.HasValue)  
{           ← Usando método  
    Console.WriteLine("i is not null");  HasValue do struct  
}           Nullable  
else  
{  
    Console.WriteLine("i is null");  
}
```

Atribuição de valor padrão

```
int j = i ?? 0;           ← Usando o operador??(null  
int j = i.GetValueOrDefault(0);  coalescing)  
int j = i.HasValue ? i.Value : 0;  ← Usando o método struct  
                                         Nullable  
                                         ← Usando o operador  
                                         condicional ?:
```

Membros da struct Nullable

```
.HasValue // false if null, true if valid value  
.Value // gets underlying value  
.GetValueOrDefault() // underlying value or default  
.GetValueOrDefault(default) // value or specified default
```

Programação Orientada a Objetos



Considerações com Structs



- As Structs geralmente são atribuídas na pilha (Stack) em vez de no Heap. Isso permite uma alocação de memória mais rápida, já que as operações de pilha geralmente são mais eficientes.
- Geralmente são menores em comparação com as classes.
- Eles são comparados e atribuídos com base em seus valores reais, em vez de referências.
- Eles não podem herdar diretamente de outras estruturas ou classes. No entanto, eles podem implementar interfaces para fornecer um nível de abstração e reutilização de código.
- Eles podem ser usados em alguns cenários sensíveis ao desempenho, o uso de estruturas pode fornecer benefícios de desempenho devido ao seu mapeamento de pilha e semântica de valor.
- Eles também podem ser usados se a imutabilidade for necessária, desde que haja apenas campos ou propriedades readonly.

Programação Orientada a Objetos



Dados por valor e por referência usando classes e structs

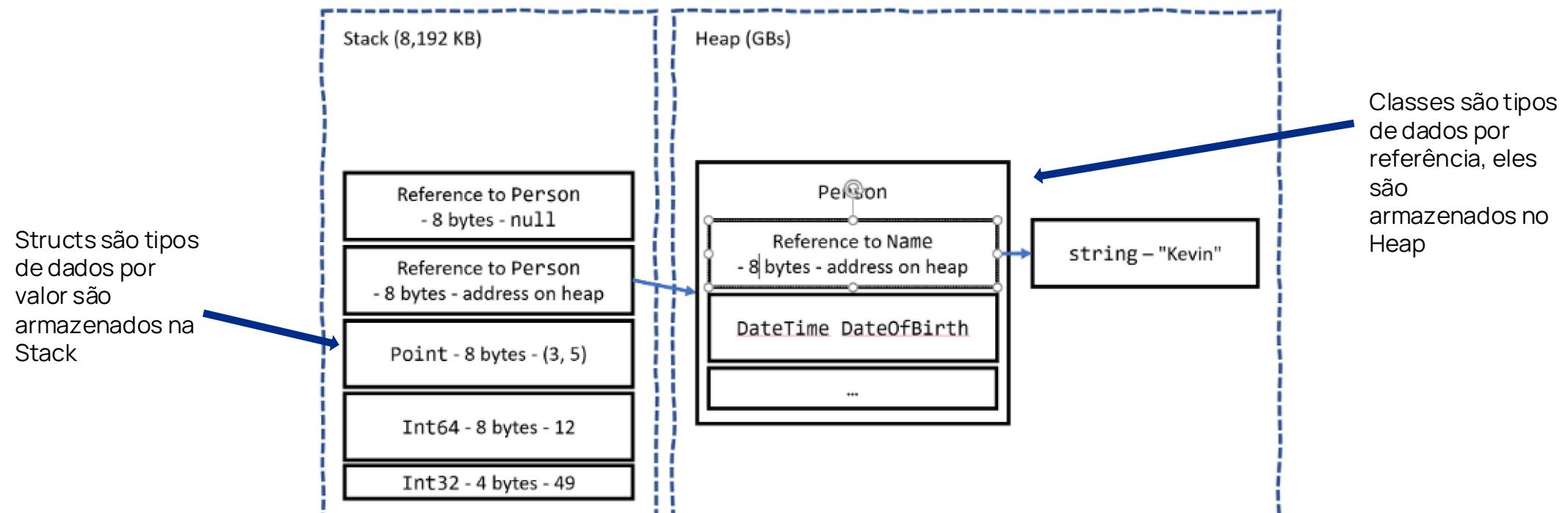


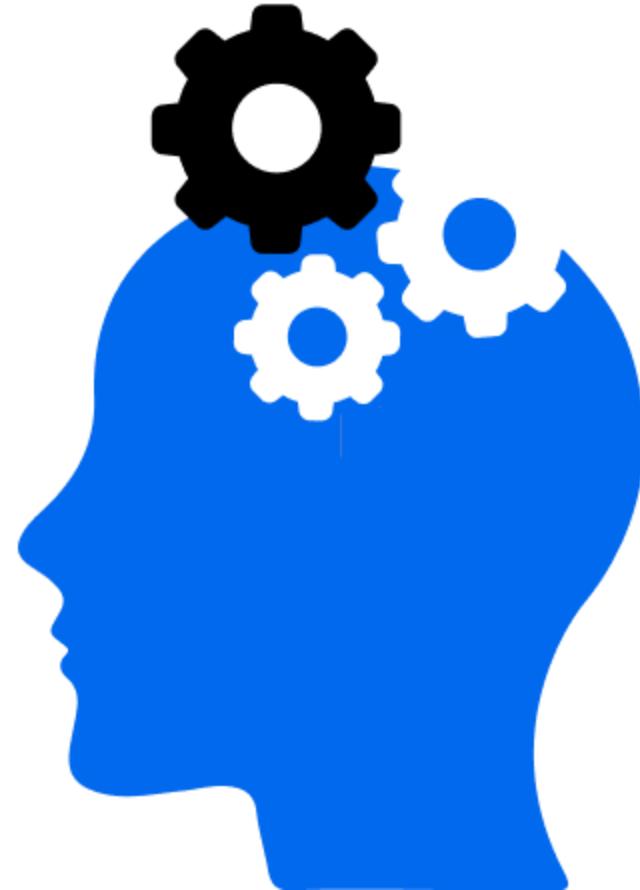
Figure 6.1: How value and reference types are allocated in the stack and heap

Programação Orientada a Objetos

Discussão Rápida (5min)

Atividade em Grupo

Indique quais são os pilares da
Programação Orientada a Objetos.



Programação Orientada a Objetos



Algumas diferenças entre Java e C#

Característica	C#	Java
Sobreescrita de Métodos	Se desejar que um método em uma classe derivada substitua um método na classe base, você deverá usar a palavra-chave <i>override</i> .	Os métodos são virtuais por padrão, o que significa que eles podem ser substituídos em uma subclasse.
Sobrecarga de operadores	Você pode redefinir ou sobrecarregar a maioria dos operadores internos disponíveis em C#.	Não suporta sobrecarga do operador
Delegates	Recurso nativo, eles são semelhantes aos ponteiros de função em C++. Os <i>delegates</i> permitem que você passe métodos como argumentos para outros métodos e atribui-los a variáveis.	Ele não tem um equivalente direto aos <i>delegates</i> , mas existem algumas soluções que podem ser usadas para alcançar resultados semelhantes.
Nullable types	Ele tem suporte interno para tipos de aceitação <i>null</i> , que são variáveis que podem ser <i>null</i> ou ter um valor.	Se você quiser criar um tipo de aceitação nula que também possa indicar se ele tem um valor ou não, talvez seja necessário criar uma classe wrapper.

Abordaremos os *delegates* extensivamente, em uma aula futura

Programação Orientada a Objetos



Algumas semelhanças entre Java e C#

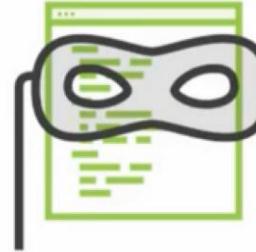
C#	Java
Ambas as linguagens suportam os quatro pilares da programação orientada a objetos: encapsulamento, herança, polimorfismo e abstração.	
Ambas as linguagens têm classes e objetos como componentes fundamentais do código.	
Ambos os idiomas oferecem suporte a modificadores de acesso (público, privado, protegido e interno) para controlar o acesso aos membros da classe.	
Ambos os idiomas oferecem suporte a substituição e substituição de métodos.	
Ambas as linguagens suportam interfaces para definir contratos abstratos que podem ser implementados usando classes.	

Em geral, Java e C# são poderosas linguagens OOP, bastante semelhantes

Programação Orientada a Objetos

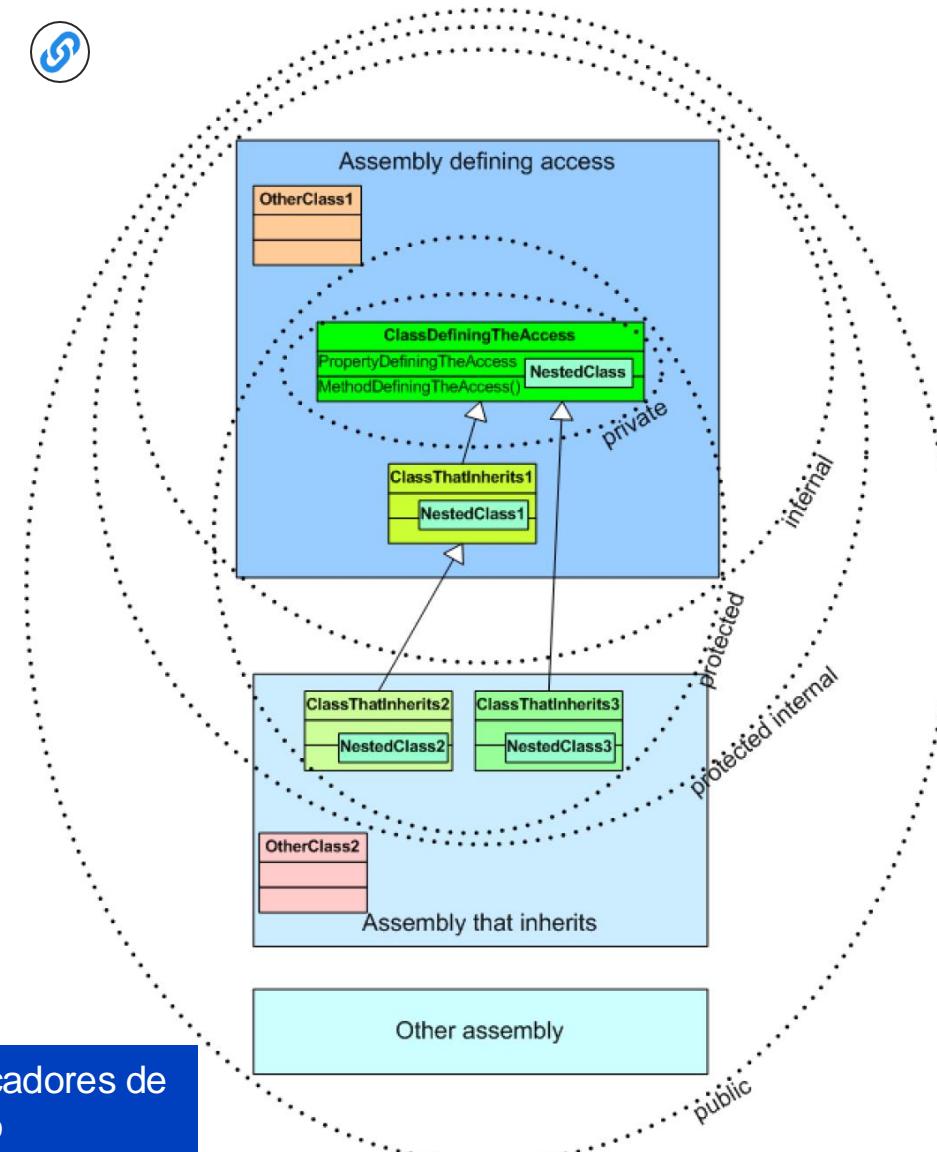


Encapsulamento



- Protege os estados internos de um objeto usando modificadores de acesso e permitindo que ele só seja modificado por métodos controlados
- Ajuda a gerenciar a complexidade
- A implementação pode mudar sem afetar a aplicação
- Ajuda a impedir o acesso não autorizado ou modificações

Modificadores de acesso



Programação Orientada a Objetos



Límites de acceso	
<code>private</code>	Member is accessible inside the type only. This is the default.
<code>internal</code>	Member is accessible inside the type and any type in the same assembly.
<code>protected</code>	Member is accessible inside the type and any type that inherits from the type.
<code>public</code>	Member is accessible everywhere.
<code>internal</code> <code>protected</code>	Member is accessible inside the type, any type in the same assembly, and any type that inherits from the type. Equivalent to a fictional access modifier named <code>internal_or_protected</code> .
<code>private</code> <code>protected</code>	Member is accessible inside the type and any type that inherits from the type and is in the same assembly. Equivalent to a fictional access modifier named <code>internal_and_protected</code> . This combination is only available with C# 7.2 or later.

Classe selada

É usada quando você não deseja que uma classe herde de outra, na declaração você usa a palavra reservada *sealed*. Também é aplicável ao nível do método

```
using System;
namespace SealedClass {
    sealed class Animal {
    }

    // trying to inherit sealed class
    // Error Code
    class Dog : Animal {
    }

    class Program {
        static void Main (string [] args) {

            // create an object of Dog class
            Dog d1 = new Dog();
            Console.ReadLine();
        }
    }
}
```

Declaração de classe selada

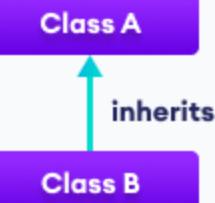
Quando tentarmos herdar dessa classe, obteremos um erro

error CS0509: 'Dog': cannot derive from sealed type 'Animal'

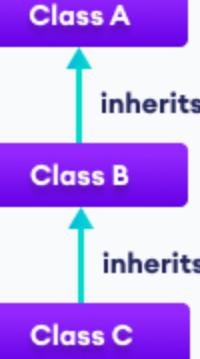
Programação Orientada a Objetos



Herança



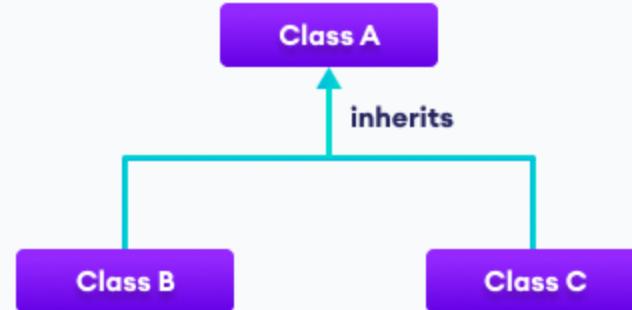
C# Single Inheritance



C# Multilevel Inheritance



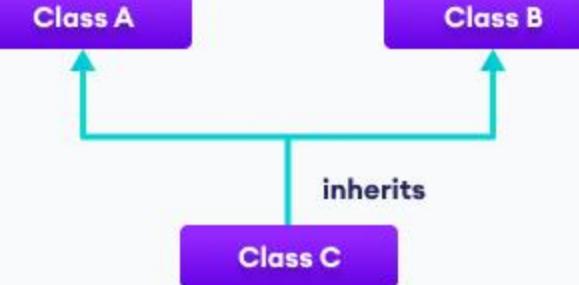
Uma classe derivada herda de uma classe base



C# Hierarchical Inheritance



Várias classes derivadas herdam de uma classe base



Multiple Inheritance



Uma classe derivada herda de várias classes base, em C# isso é suportado usando interfaces

A seguir veremos alguns exemplos

Programação Orientada a Objetos



Herança implícita

Object

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
#ctor()

Todos os tipos de dados herdam de Object, mesmo que ele não seja exibido explicitamente

Key

Unique member	
Inherited member	
Overridden member	

Publication

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
    private DateTime _datePublished;
    private int _totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The number of pages cannot be less than or equal to zero.");
            _totalPages = value;
        }
    }
}
```

Parte da Classe
Publication

Programação Orientada a Objetos



Herança Simples



Publication
Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

Key

Unique member	
Inherited member	
Overridden member	

Book
Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()



```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
        this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher) :
    base(title, publisher, PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string without "-" characters.
        // We could also determine whether the ISBN is valid by comparing its checksum digit
        // with a computed checksum.
        //
        if (!string.IsNullOrEmpty(isbn))
        {
            // Determine if ISBN length is correct.
            if (!(isbn.Length == 10 || isbn.Length == 13))
                throw new ArgumentException(
                    "The ISBN must be a 10- or 13-character numeric string.");
            if (!ulong.TryParse(isbn, out _))
                throw new ArgumentException(
                    "The ISBN can consist of numeric characters only.");
        }
        ISBN = isbn;
        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string? Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public decimal SetPrice(decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException(nameof(price),
                "The price cannot be negative.");
        decimal oldValue = Price;
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-character string.");
        Currency = currency;

        return oldValue;
    }
}
```

Book herda de Publication

Usando o construtor da classe pai usando a palavra **base**

Programação Orientada a Objetos



Herança hierárquica



```
public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;
    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

Declarão de classe abstrata

Declarão de membros abstratos

- Este exemplo usa uma classe abstrata, C# tem suporte para esse tipo de classe.
- Esses tipos de classes não podem ser instanciados e exigem a palavra *abstract* na sua declaração,
- Podem ter membros que possuem uma implementação ou não (membros *abstract*)
- Os membros *abstract* não tem corpo e deverão ser *sobrescritos* pelas classes filhas, para tal se usa a palavra reservada *override*.

Rectangle herda da Classe Shape

→

```
public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }
    public double Width { get; }
    public override double Area => Length * Width;
    public override double Perimeter => 2 * Length + 2 * Width;
    public bool IsSquare() => Length == Width;
    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}
```

Implementação (sobrescrita) de membros abstratos

Circle herda de classe Shape

→

```
public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }
}
```

Implementação (sobrescrita) de membros abstratos

→

```
public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);
public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);
// Define a circumference, since it's the more familiar term.
public double Circumference => Perimeter;
public double Radius { get; }
public double Diameter => Radius * 2;
}
```



Programação Orientada a Objetos



④ Herança de vários níveis

```
class Shape {  
    double a_width;  
    double a_length;  
    // Default constructor  
    public Shape()  
    {  
        Width = Length = 0.0;  
    }  
    // Constructor for Shape  
    public Shape(double w, double l)  
    {  
        Width = w;  
        Length = l;  
    }  
    // Construct an object with  
    // equal length and width  
    public Shape(double y)  
    {  
        Width = Length = y;  
    }  
    // Properties for Length and Width  
    public double Width  
    {  
        get { return a_width; }  
  
        set { a_width = value < 0 ? -value : value; }  
    }  
  
    public double Length  
    {  
        get { return a_length; }  
  
        set { a_length = value < 0 ? -value : value; }  
    }  
    public void DisplayDim()  
    {  
        Console.WriteLine("Width and Length are " +  
                         + Width + " and " + Length);  
    }  
}
```

Classe Base

```
class Rectangle : Shape {  
  
    string Style;  
    // A default constructor.  
    // This invokes the default  
    // constructor of Shape.  
    public Rectangle()  
    {  
        Style = "null";  
    }  
    // Constructor  
    public Rectangle(string s, double w, double l)  
        : base(w, l)  
    {  
        Style = s;  
    }  
    // Construct an square.  
    public Rectangle(double y)  
        : base(y)  
    {  
        Style = "square";  
    }  
    // Return area of rectangle.  
    public double Area()  
    {  
        return Width * Length;  
    }  
    // Display a rectangle's style.  
    public void DisplayStyle()  
    {  
        Console.WriteLine("Rectangle is " + Style);  
    }  
}
```

Classe derivada de Shape

```
class ColorRectangle : Rectangle {  
  
    string rcolor;  
  
    // Constructor  
    public ColorRectangle(string c, string s,  
                          double w, double l)  
        : base(s, w, l)  
    {  
        rcolor = c;  
    }  
  
    // Display the color.  
    public void DisplayColor()  
    {  
        Console.WriteLine("Color is " + rcolor);  
    }  
}
```

Classe derivada de Rectangle

Programação Orientada a Objetos



Interfaces e herança múltipla

```
interface IPolygon {  
    // method without body  
    void calculateArea();  
}
```

Declaração de interface

Assinatura de interface,
não é possível usar
modificadores de acesso

- A declaração requer o uso da palavra reservada Interface.
- É comum que o nome da interface comece com a letra I.
- Os membros de uma interface não têm implementação. Embora em versões mais recentes do C# eles possam ter uma implementação padrão.
- Eles podem conter os seguintes membros: propriedades, métodos, eventos, indexadores.
- Os membros são automaticamente públicos

```
interface IPolygon {  
    // method without body  
    void calculateArea(int a, int b);  
}  
  
interface IColor {  
    void getColor();  
}  
  
// implements two interfaces  
class Rectangle : IPolygon, IColor {  
  
    // implementation of IPolygon interface  
    public void calculateArea(int a, int b) {  
        int area = a * b;  
        Console.WriteLine("Area of Rectangle: " + area);  
    }  
  
    // implementation of IColor interface  
    public void getColor() {  
        Console.WriteLine("Red Rectangle");  
    }  
}  
  
class Program {  
    static void Main (string [] args) {  
  
        Rectangle r1 = new Rectangle();  
  
        r1.calculateArea(100, 200);  
        r1.getColor();  
    }  
}
```

Declaração de interfaces

Implementação de ambas as interfaces

Implementação de assinatura de interface IPolygon

Implementação da assinatura da interface IColor

Resultado

```
Area of Rectangle: 20000  
Red Rectangle
```

Programação Orientada a Objetos



Polimorfismo por Sobrecarga de Método

```
void display(int a) {  
    ...  
}  
...  
void display(int a, int b) {  
    ...  
}
```

Sobrecarregar alterando
o número de parâmetros

```
void display(int a) {  
    ...  
}  
...  
void display(string b) {  
    ...  
}
```

Sobrecarga alterando o
tipo de dados do
parâmetro

```
void display(int a, string b) {  
    ...  
}  
...  
void display(string b, int a) {  
    ...  
}
```

Sobrecarregando alterando a
ordem dos parâmetros

Os tipos de retorno
não afetam a
sobrecarga do
método.

```
public class MathUtils {  
    ...  
    public static int Add(int a, int b) {  
        ...  
        return a + b;  
    }  
    ...  
    public static double Add(double a, double b) {  
        ...  
        return a + b;  
    }  
    ...  
    int result1 = MathUtils.Add(5, 3);  
    double result2 = MathUtils.Add(2.5, 4.7);  
    ...
```

Método Add

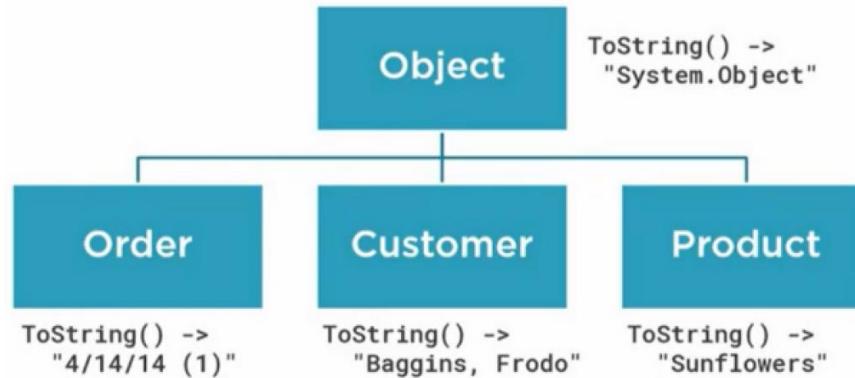
Sobrecarga de
Método Add

Uso de ambos
métodos

Programação Orientada a Objetos



Polimorfismo por sobrescrita



- A palavra-chave *virtual* é usada para modificar um método, propriedade, indexador ou declaração de evento e permitir que ele seja substituído em uma classe derivada.
- Por padrão, os métodos não são virtuais
- *virtual* não pode ser usado com modificadores *static*, *abstract*, *private*, ou *override*.
- O modificador *override* é necessário para modificar a implementação abstrata ou virtual de um membro herdado.

```
class Shape
{
    protected int width;
    protected int height;

    public Shape(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    public virtual void DisplayArea()
    {
        Console.WriteLine("Area is not defined for the base shape.");
    }
}

class Rectangle : Shape
{
    public Rectangle(int width, int height)
        : base(width, height) // Calling base class constructor
    {
    }

    public override void DisplayArea()
    {
        int area = width * height;
        Console.WriteLine($"Area of the rectangle is: {area}");
    }
}
```

Usando campos protected

Declarando o método usando virtual para habilitar sobreescrita

Sobreescrivendo o comportamento da classe base

Programação Orientada a Objetos



Outras considerações entre substituição e sobrecarga

 Sobrecarga	Sobrescrita
Envolve o uso de métodos com o mesmo nome, mas assinatura diferente (número e tipo de parâmetros)	Envolve o uso de métodos com o mesmo nome e assinatura, mas em uma estrutura diferente.
Acontece dentro de uma classe e não requer herança	Requer duas classes e Herança está envolvida
O tipo de retorno pode ser o mesmo ou diferente	A tipo de retorno deve ser o mesmo
Este é um exemplo de polimorfismo em tempo de compilação	É um exemplo de polimorfismo de tempo de execução

Outras considerações entre virtual e abstract

abstract	virtual
É um membro que não tem uma implantação	É um método com uma implementação padrão
Seu uso é limitado a classes abstratas	Pode ser usado em classes abstratas ou concretas
Deve ser substituído por classes derivadas	A sobrescrita é opcional

Programação Orientada a Objetos



Extension Methods

- Os métodos de extensão foram introduzidos no C# 3.0.
- Eles estendem e adicionam comportamento aos tipos existentes, sem criar um novo tipo derivado, recompilar ou modificar o tipo original.
- Eles são especialmente úteis quando você não pode modificar a fonte de um tipo que deseja melhorar.
- O método de extensão pode ser invocado como se fosse um método de membro do tipo original. Isso permite o encadeamento de métodos usados para implementar uma interface fluída.
- Os métodos de extensão usam um primeiro parâmetro especial que designa o tipo original que está sendo estendido.
- O parâmetro é decorado com a palavra-chave *this* (que constitui uma utilização especial e distinta do *this* convencional, O convencional permite referência a outros membros dentro da classe)

Programação Orientada a Objetos



Extension Methods

Declaração e uso

Por convenção, qualquer classe estática cuja finalidade é definir métodos de extensão tem um nome que termina com a palavra Extensions e começa com o nome do tipo de dados a ser estendido

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

O tipo de dados a ser estendido é indicado

Se necessário, é possível definir outros parâmetros para o método

Eles também podem definir um tipo específico de retorno

```
// This calls the extension method StringExtensions.Shorten()
var newString = myString.Shorten(5);

"some string".Shorten(5);
```

Usando o método de extensão definida

Programación Orientada a Objetos



Extension Methods

Exemplo com strings

```
public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }

    string nullString = null;
    string emptyString = nullString.EmptyIfNull(); // will return ""
    string anotherNullString = emptyString.NullIfEmpty(); // will return null
}
```

Declarando o método de extensão EmptyIfNull

Declarando o método de extensão NullIfEmpty

Usando métodos de extensão

Exemplo con Enums

```
public enum YesNo
{
    Yes,
    No,
}

public static class EnumExtentions
{
    public static bool ToBool(this YesNo yn)
    {
        return yn == YesNo.Yes;
    }

    public static YesNo ToYesNo(this bool yn)
    {
        return yn ? YesNo.Yes : YesNo.No;
    }
}
```

Declaração do Enum

Declaração de métodos de extensão para o Enum criado

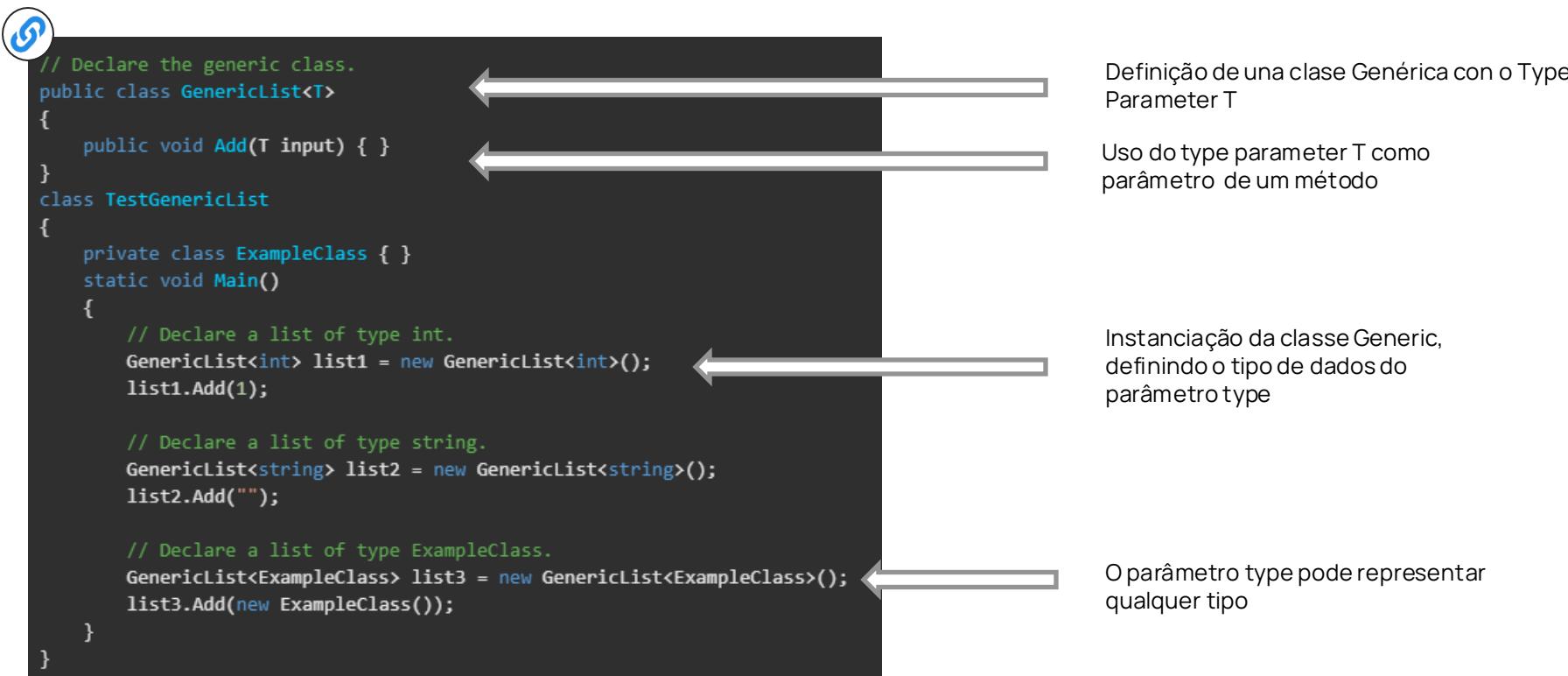
Genéricos



Genéricos

Genéricos

Os genéricos introduzem o conceito de parâmetros de tipo no .NET, permitindo que você crie classes e métodos que adiam a especificação de um ou mais tipos até que o código do cliente declare e instancie a classe ou o método.





Genéricos

Genéricos e Inferência de Tipo

Usando inferência de tipo do compilador C# não é necessário especificar o tipo de dados, no entanto, isso não se aplica a construtores

```
class Tuple<T1, T2>
{
    public Tuple(T1 value1, T2 value2)
    {
    }
}

var x = new Tuple(2, "two");
var y = new Tuple<int, string>(2, "two");
```

Definindo uma classe genérica com dois Type Parameters

Definindo seu construtor

Esta linha de código no funcionará

Os tipos de dados precisam ser explicitamente definidos

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
        return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two"); // This WILL work...
```

Definindo um método genérico

Esse método instancia o objeto anterior

A inferência de tipos nos ajuda, nesse caso, a não especificar explicitamente os tipos de dados

Genéricos



Constraints

As restrições (constraints) informam ao compilador sobre os recursos que um argumento de tipo deve ter (valor do Type parameter). Sem restrições, o argumento type pode ser de qualquer tipo. Aqui estão alguns exemplos de como aplicar restrições.

```
class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}
```

Definição da constraint usando a palavra-chave *where*

Como a restrição indica que o parâmetro type tem um construtor vazio, esse construtor é usado para uma nova instância do tipo T

```
interface IType;
interface IAnotherType;

// T must be a subtype of IType
interface IGeneric<T>
    where T : IType
{
```

← Declaração de interfaces

← Definição da restrição usando a palavra-chave *where*, indica que o parâmetro Type deve ser uma implementação IType

```
class Type : IType { }
class Sub : IGeneric<Type> { }
```

← Implementação de interface, tipo **deve** implementar IType

A restrição `new()` é usada para restringir argumentos de tipo de ter um construtor vazio por padrão. (Um construtor sem parâmetros)

As interfaces também podem ser usadas como restrições, portanto, os argumentos de tipo devem ser implementações da interface



Genéricos

Constraints

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.  
// Interfaces are valid, as they are reference types  
class AcceptsRefType<TRef>  
    where TRef : class  
{  
    // TStruct must be a value type.  
    public void AcceptStruct<TStruct>()  
        where TStruct : struct  
    {  
    }  
  
    // If multiple constraints are used along with class/struct  
    // then the class or struct constraint MUST be specified first  
    public void Foo<TComparableClass>()  
        where TComparableClass : class, IComparable  
    {  
    }  
}
```

Definindo restrição, restringindo o valor a ser um tipo por referência

Definição de restrição, restringindo o valor a ser um tipo por valor

Definição de restrição, restringindo que o valor é um tipo por referência que implementa IComparable

Também é possível especificar se o argumento type deve ser um tipo por referência ou por valor usando as restrições class ou struct.

```
class Generic<T, T1>
```

```
    where T : IType  
    where T1 : Base, new()
```

```
{  
}
```

Definindo uma restrição para T

Definindo restrição para T1

É possível usar restrições diferentes se houver dois ou mais parâmetros de tipo, ou seja, uma restrição específica pode ser aplicada para cada um. Também é possível combinar restrições, embora existam algumas combinações que não podem ser usadas.

Genéricos



Outras constraints existentes



Constraint	Descripción
where T : class?	O argumento type deve ser um tipo de referência, independentemente de aceitar ou não valores NULL.
where T: notnull	O argumento type deve ser um tipo que não aceita valoresNULL.
where T :< nombre de clase base >	O argumento type deve ser ou derivado da classe base especificada.
where T :< nombre de clase base >?	O argumento type deve ser ou derivado da classe base especificada. Em um contexto que aceita valores NUL
where T : U	O argumento do tipo fornecido por T deve ser ou deve ser derivado do argumento fornecido para U



Genéricos

Extension methods genéricos

```
class MyType<T1, T2>
{
}

static class Extensions
{
    public static void Example<T>(this MyType<int, T> test)
    {
    }
}
```



Declaração de classe com dois type parameters



Extension method, definindo a T1 como int e o type parameter T como argumento de tipo T2

Calling it would be like:

```
MyType<int, string> t = new MyType<int, string>();
t.Example();
```



Uso do Extension method



Genéricos

Extension methods genéricos

Declaração de um método de extensão genérico usando restrições. Use o valor padrão do objeto EqualityComparer e compare-o com o valor padrão do objeto de parâmetro type

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>
{
    return EqualityComparer<T>.Default.Equals(obj, default(T));
}
```

Calling code:

Uso do Extension method

```
int number = 5;
var IsDefault = number.IsDefault();
```

Genéricos



Uso de genéricos

O uso de genéricos ocorre em diferentes ocasiões, entre seus usos mais comuns estão estruturas de dados ou coleções, que veremos na próxima seção, também existem interfaces que cuidam de tarefas comuns que ajudam em tarefas de ordenação ou gerenciamento de recursos:

Interface	Método	Descripción
IComparable	CompareTo(outro)	Define um método de comparação que um tipo de dados implementa para classificar suas instâncias
IComparer	Compare(primário, secundário)	Define um método de comparação que um tipo de dados filho implementa para classificar instâncias do tipo de dados pai
IDisposable	Dispose()	Define um método que ajuda a liberar recursos não manipulados com mais eficiência do que usar o finalizador



Genéricos

Comparações

IComparable

```
Person[] people =  
{  
    new() { Name = "Simon" },  
    new() { Name = "Jenny" },  
    new() { Name = "Adam" },  
    new() { Name = "Richard" }  
};  
WriteLine("Initial list of people:");  
foreach (Person p in people)  
{  
    WriteLine($" {p.Name}");  
}  
WriteLine("Use Person's IComparable  
implementation to sort:");  
Array.Sort(people);
```

Unhandled Exception:
System.InvalidOperationException:
Failed to compare two elements in the
array. ---> System.ArgumentException:
At least one object must implement
IComparable.

Criamos um array de
objetos Person

Imprimimos seus nomes

Usamos o método sort
pertencente à classe Array, que
usa a lógica de comparação dada
por um objeto que implementa
IComparable

Isso nos dará uma exceção
porque Person não tem
uma definição de como
fazer comparações

```
public class Person : object,  
IComparable<Person>
```

```
public int CompareTo(Person? other)  
{  
    if (Name is null) return 0;  
    return Name.CompareTo(other?.Name);  
}
```

Initial list of people:

Simon
Jenny
Adam
Richard

Use Person's IComparable implementation
to sort:

Adam
Jenny
Richard
Simon

Person agora implementa
IComparable de Person

Implementação do método
de comparação. Os nomes
estão sendo usados para
fazer a comparação

Ao executar o código, você
vê a lista inicial e a lista
ordenada



Genéricos

IComparer

Útil quando não há acesso ao código-fonte de um tipo. Você pode criar um tipo separado que implementa a interface IComparer

```
namespace Packt.Shared;
public class PersonComparer : IComparer<Person>
{
    public int Compare(Person? x, Person? y)
    {
        if (x is null || y is null)
        {
            return 0;
        }
        // Compare the Name lengths...
        int result = x.Name.Length.CompareTo(y.Name.Length);
        // ...if they are equal...
        if (result == 0)
        {
            // ...then compare by the Names...
            return x.Name.CompareTo(y.Name);
        }
        else // result will be -1 or 1
        {
            // ...otherwise compare by the lengths.
            return result;
        }
    }
}
```

Implementação da interface IComparer

Implementação de assinatura,
O objetivo é definir o algoritmo de comparação para o objeto Person.
Retorna 0 se for igual e +1 ou -1 se houver variações nas posições dos caracteres

Nesse caso, ao usar o método Sort, é necessário instanciar o objeto que implementa IComparer (PersonComparer)

```
Array.Sort(people, new PersonComparer());
foreach (Person p in people)
{
    WriteLine($" {p.Name}");
}
```

Resultado

Use PersonComparer's IComparer implementation to sort:

Adam
Jenny
Simon
Richard

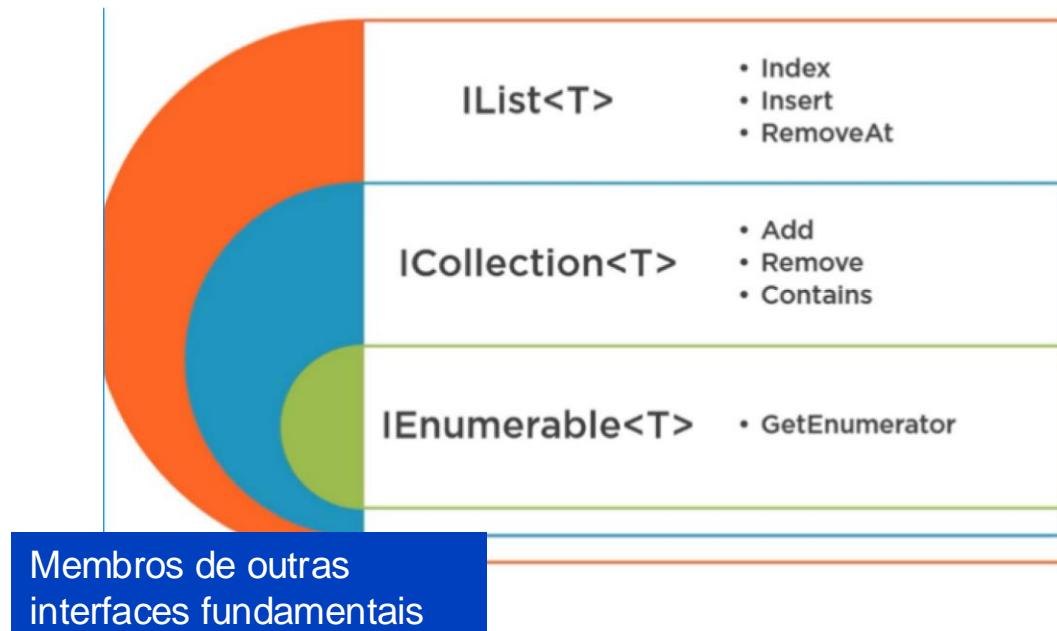
Arrays, Listas e Dicionários

Arrays, Listas y Diccionarios



Interfaces comuns para coleções

No contexto de coleções, as interfaces são usadas para definir um conjunto de operações comuns que podem ser executadas por qualquer coleção. Entre as fundamentais estão: `IEnumerable`, `IEnumerable`, `ICollection` e `IList`



`IEnumerable<T>` é usado para definir a lógica de iteração na coleção.
`IEnumerable`: fornece métodos para iterar sobre uma coleção de itens. Defina propriedades como *Current* para obter o item atual na coleção e métodos como *MoveNext()* para mover para o próximo item e *Reset()* para redefinir o enumerador.



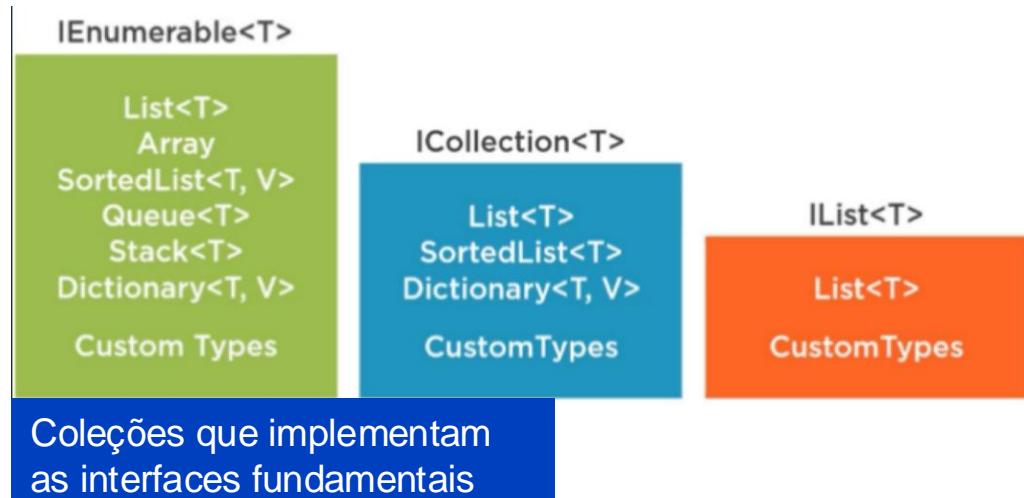
Arrays, Listas y Diccionarios

Interfaces comuns para coleções

`IEnumerable<T>`: Essa interface define uma coleção que pode ser iterada.

`ICollection<T>`: Essa interface define as operações básicas de uma coleção, como adicionar, remover e consultar itens.

`<T> IList`: Essa interface estende o `ICollection` para `<T>` fornecer operações adicionais para acessar itens em uma coleção por índice.



Namespace	Example type(s)	Description
<code>System .Collections</code>	<code>IEnumerable</code> , <code>IEnumerable<T></code>	Interfaces and base classes used by collections.
<code>System .Collections .Generic</code>	<code>List<T></code> , <code>Dictionary<T></code> , <code>Queue<T></code> , <code>Stack<T></code>	Introduced in C# 2.0 with .NET Framework 2.0. These collections allow you to specify the type you want to store using a generic type parameter (which is safer, faster, and more efficient).

Arrays, Listas y Diccionarios



Interfaces comuns para coleções

```
namespace System.Collections
{
    public interface IEnumerable
    {
        Ienumerator GetEnumerator();
    }
}

namespace System.Collections
{
    public interface Ienumerator
    {
        object Current { get; }
        bool MoveNext();
        void Reset();
    }
}
```

Assinatura de
IEnumerable

```
namespace System.Collections
{
    public interface ICollection : IEnumerable
    {
        int Count { get; }
        bool IsSynchronized { get; }
        object SyncRoot { get; }
        void CopyTo(Array array, int index);
    }
}

namespace System.Collections.Generic
{
    [DefaultMember("Item")] // aka this indexer
    public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
    {
        T this[int index] { get; set; }
        int IndexOf(T item);
        void Insert(int index, T item);
        void RemoveAt(int index);
    }
}
```

Assinatura de ICollection,
herda de IEnumerable

Assinatura de IList, herda
de ICollection

Fundamentos de C#



Arrays

```
// create an array with 2 elements
var myArray = new [] { "one", "two" }; ← Declarando e inicializando o Array

// enumerate through the array
foreach(var item in myArray) ← Iteração usando foreach
{
    Console.WriteLine(item);
}

// output:
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else"; ← Acesso ao primeiro elemento usando index

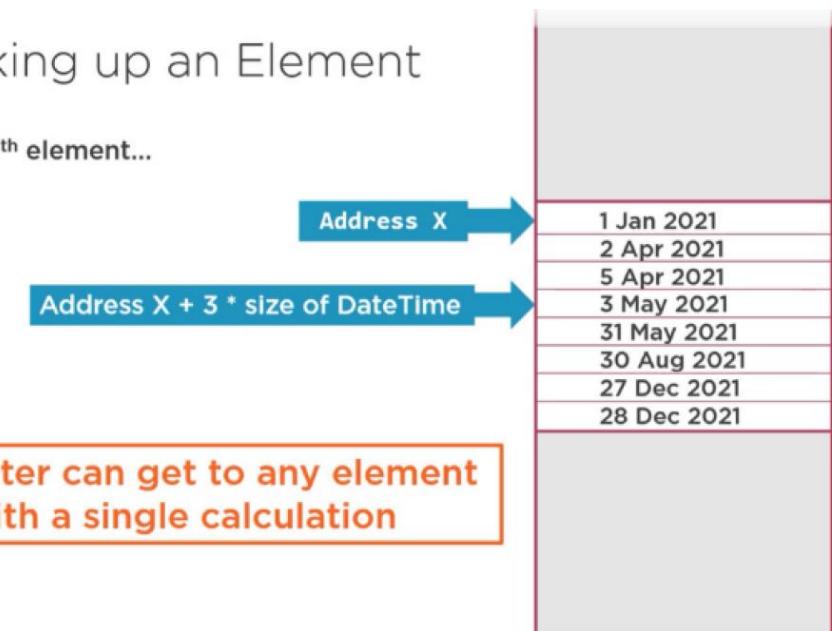
// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else ← Resultado
```

Quão rápido é pesquisar uma matriz usando o índice

Looking up an Element

To get 4th element...



Fundamentos de C#



Arrays

Array multidimensional

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };  
  
// Access a member of the multi-dimensional array:  
Console.WriteLine(arr[3, 1]); // 4
```

Declarando e inicializando a matriz de 2 dimensões, separada por vírgula

Obter o item na linha e coluna especificadas

Jagged Array

Os jagged array são arrays que, ao invés de tipos primitivos, contém arrays (ou outras coleções). É como uma série de arrays, cada elemento contém outro array.

```
int[][] a = new int[8][];
```

Declaração do Array

The second [] is initialized without a number. To initialize the sub arrays, you would need to do that separately:

```
for (int i = 0; i < a.length; i++)  
{  
    a[i] = new int[10];  
}
```

a[<row_number>][<column_number>]

Definindo valores dos elementos do array, neste caso novos arrays que tenham 10 elementos

Asignar un array a la interfaz IEnumerable

Arrays implementan IEnumerable es por esa razón que es posible asignar un array a una variable con ese tipo de dato

```
int[] arr1 = { 3, 5, 7 };  
IEnumerable<int> enumerableIntegers = arr1;  
List<int> listOfIntegers = new List<int>();  
listOfIntegers.AddRange(arr1); // You can pass
```

Criando uma lista a partir de um array

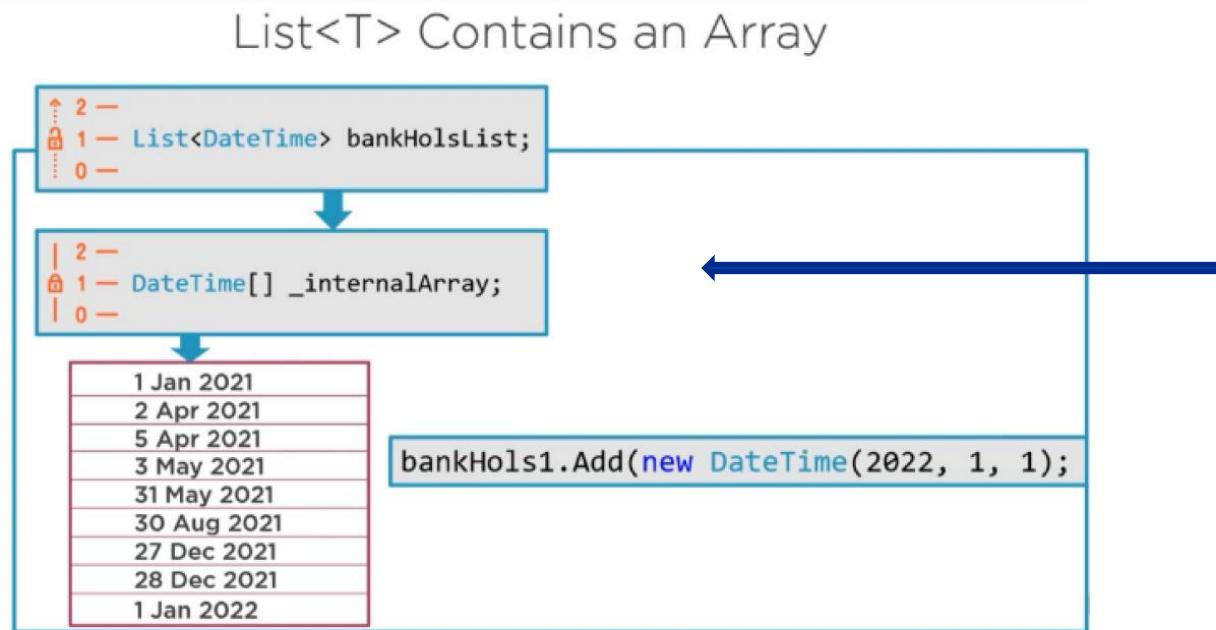
São recomendados quando você tem um número fixo de itens que você pode acessar por meio de um índice

Fundamentos de C#



Listas

É uma coleção genérica de objetos de um tipo específico de tamanho dinâmico. Usando esta coleção, podemos adicionar, inserir, excluir e pesquisar por índice. `List<T>` pode ser pensada como um Array cujo tamanho pode mudar. Eles ocupam muito mais memória do que uma matriz justamente por causa de sua capacidade de crescer dinamicamente.



As listas contêm um array internamente que tem um número específico de itens extras para poder adicionar rapidamente

Quando esse limite é excedido, um novo array é gerado internamente

Fundamentos de C#



Listas

```
using System.Collections.Generic; ← Usando o namespace Collections  
  
var list = new List<int>() { 1, 2, 3, 4, 5 }; ← Declarando e atribuindo uma lista int  
list.Add(6); ← Adicionando um elemento  
Console.WriteLine(list.Count); // 6 ← Usando a propriedade Count na lista  
list.RemoveAt(3); ← Removendo o item no índice 3  
Console.WriteLine(list.Count); // 5  
Console.WriteLine(list[3]); // 5 ← Obtendo o item de índice 3 usando o operador do indexador []
```

```
var numbers = new List<int>(){ 10, 20, 30, 40 }; // Declarando outra lista  
  
numbers.Insert(1, 11); // Inserir o número 11 no índice 1, ou seja, após 10  
  
foreach (var num in numbers) {  
    Console.WriteLine(num);  
}  
  
numbers.Contains(10); // returns true  
numbers.Contains(11); // returns false  
numbers.Contains(20); // returns true // Verificando se a lista contém um item de valor específico
```

Fundamentos de C#



Listas

```
var numbers = new List<int>(){ 10, 20, 30, 40, 10 };           ← Declaração de lista  
  
numbers.Remove(10); // removes the first 10 from a list          ← Removendo elementos por seu valor  
  
numbers.RemoveAt(2); //removes the 3rd element (index starts from 0) ← Eles também podem ser removidos usando o índice,  
                                                               caso em que essa operação pode causar uma exceção  
                                                               porque esse índice não existe
```

Listas con objetos

```
List<Person> peopleList = new List<Person>();           ← Declaração de Lista de Objetos Person  
  
peopleList.Add(new Person { Name = "Alice", Age = 30 });  
peopleList.Add(new Person { Name = "Bob", Age = 25 });  
peopleList.Add(new Person { Name = "Charlie", Age = 35 });  
peopleList[0].Age = 32;           ← Adicionando objetos na lista  
Console.WriteLine("People in the list:");  
foreach (var person in peopleList)  
{  
    Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");  
}  
  
class Person           ← Classe Person  
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
}
```

Classe Person

Eles são recomendados quando você está lidando com coleções de objetos que podem mudar dinamicamente e precisam fazer operações CRUD

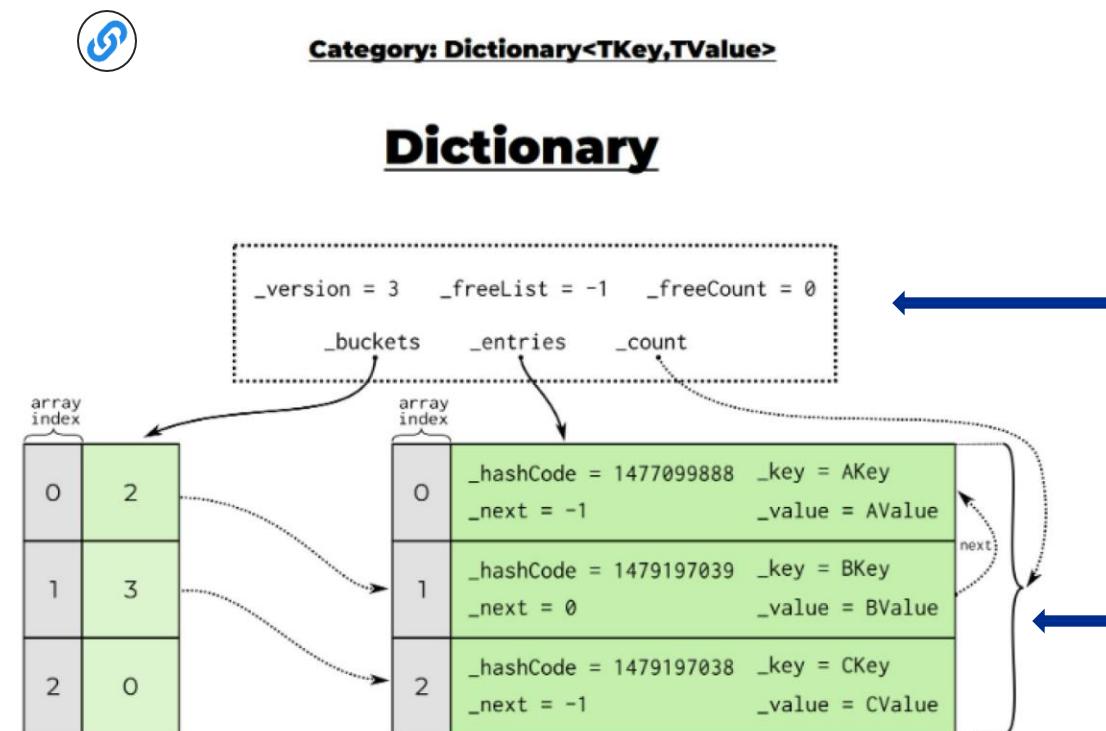
Fundamentos de C#



Dicionários

É uma coleção de pares key-value que fornecem uma pesquisa rápida usando as chaves ou *keys*. Um dicionário C# usa uma estrutura chamada *hash table* para armazenar key-value pairs.

A tabela de hash é como uma grande lista que contém vários buckets. Cada bucket armazena itens com o mesmo código hash. Quando adicionamos um item, o código hash da chave é calculado para encontrar o bucket correto. Dentro do bucket, os itens são armazenados como uma lista vinculada. Cada item tem um índice e um ponteiro para o próximo item da lista. Isso permite acesso rápido e eficiente aos itens do dicionário.



Version: alterações que modificam o Dicionário

Buckets : Elementos com hashes semelhantes

Entries: Elementos do dicionário

Count: Número de itens atuais no dicionário

Key: a chave para identificar o elemento é do tipo TKey

Value: valor do tipo TValue

HashCode: valor numérico usado para identificar um objeto

Fundamentos de C#



Dicionários

```
using System.Collections.Generic;
var people = new Dictionary<string, int>
{
    { "John", 30 }, { "Mary", 35}, {"Jack", 40}
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException

int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// Adding and changing data
people["John"] = 40;    // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

Usando o namespace de Collections

Declarando e atribuindo valores a um dicionário

Ao imprimir valores, você usa as chaves para acessar os itens.
Se eles não existirem, uma exceção será lançada

Usando TryGetValue para tentar obter um valor com a chave especificada

Atualizando um valor usando a chave
Adicionando um elemento

Iterando no dicionário e exibindo a chave e o valor no console

Iterando na coleção Keys no dicionário

Iterando na coleção de valores do dicionário

Fundamentos de C#



Dicionários

```
Dictionary<int, Person> peopleDictionary = new Dictionary<int, Person>(); ← Declaração do Dicionário com valores do tipo Person
peopleDictionary.Add(1, new Person { ID = 1, Name = "Alice" });
peopleDictionary.Add(2, new Person { ID = 2, Name = "Bob" });
peopleDictionary.Add(3, new Person { ID = 3, Name = "Charlie" });
int searchID = 2;
if (peopleDictionary.ContainsKey(searchID)) ← Adicionando valores ao dicionário
{
    Person person = peopleDictionary[searchID];
    Console.WriteLine($"Person with ID {searchID}: Name: {person.Name}");
}
else
{
    Console.WriteLine($"Person with ID {searchID} not found.");
}

Console.WriteLine("\nPeople in the dictionary:");
foreach (var pair in peopleDictionary) ← Usando o método ContainsKey do objeto Dictionary para verificar se a chave existe
{
    Console.WriteLine($"ID: {pair.Key}, Name: {pair.Value.Name}");
}

if (peopleDictionary.ContainsKey(1))
{
    peopleDictionary[1].Name = "Alicia"; ← Iterando sobre o dicionário e imprimindo a propriedade Name dos objetos Person
}

peopleDictionary.Remove(3); ← Modificando o valor da propriedade Name do elemento person associado à chave 1.
peopleDictionary.Clear(); ← Removendo el elemento con el key 3
                            Eliminando todos los elementos
```

```
class Person
{
    public int ID { get; set; }
    public string Name { get; set; }
}
```

Clase Person

Declaração do Dicionário com valores do tipo Person

Adicionando valores ao dicionário

Usando o método ContainsKey do objeto Dictionary para verificar se a chave existe

Iterando sobre o dicionário e imprimindo a propriedade Name dos objetos Person

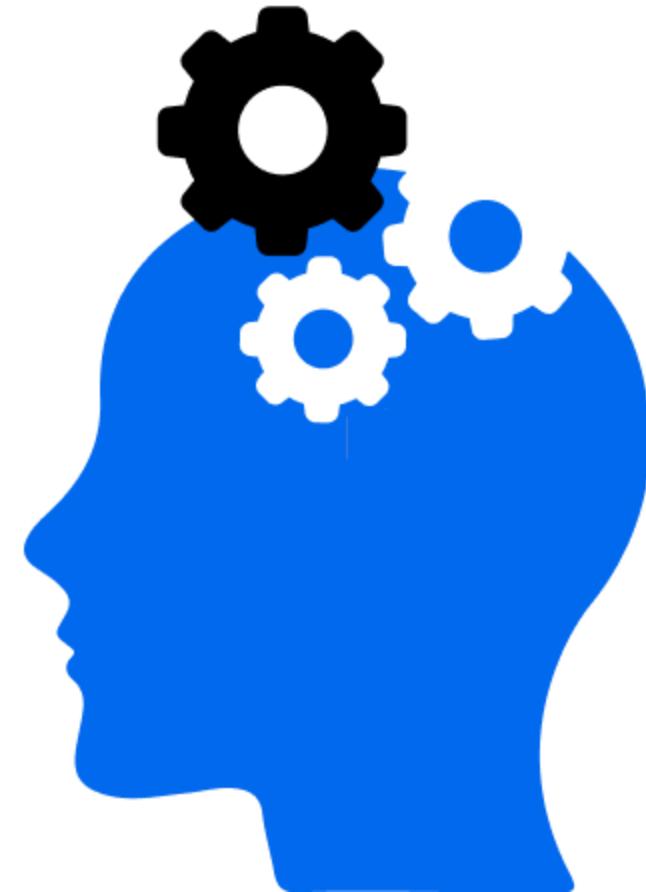
Modificando o valor da propriedade Name do elemento person associado à chave 1.

Eles são recomendados quando você está lidando com coleções que podem usar pares key-value e você precisa fazer operações CRUD enfatizando leituras rápidas

Discussão Rápida (7min)

Atividade em pares

Descreve as diferenças entre estruturas de dados em Java e C# (leva em conta Listas, Matrizes e Dicionários)



Conclusões

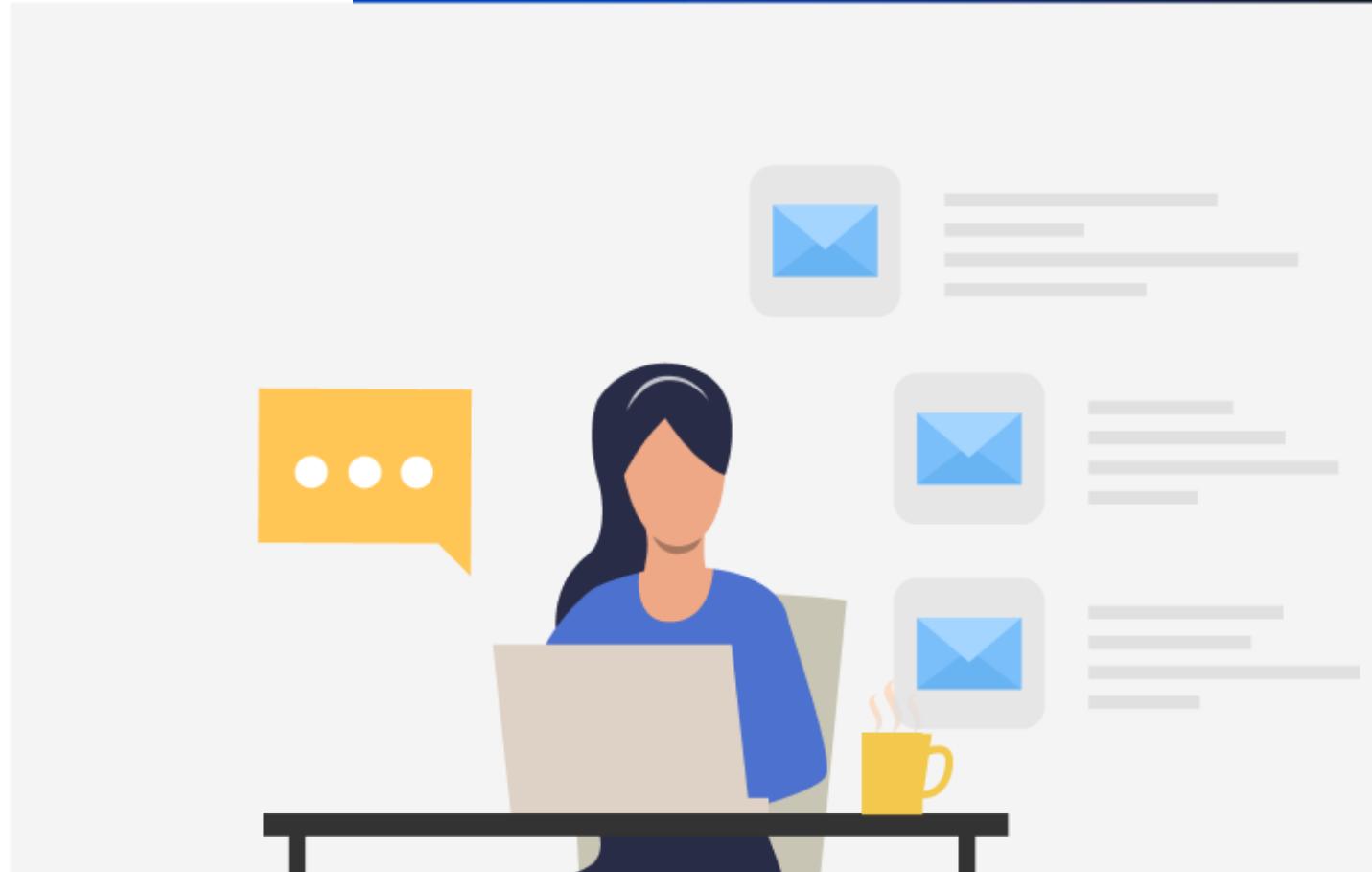
- As classes podem ter os seguintes membros: propriedades, campos, construtores, destruidores, eventos, métodos, indexadores e tipos aninhados.
- O Struct é um tipo de dados complexo que pode agrupar membros diferentes e é um tipo de dados por valor.
- C# é uma linguagem orientada a objetos e tem alguns recursos que melhoraram seu suporte
 - Interfaces, classes abstratas, herança, polimorfismo, modificadores de acesso, classes seladas e métodos de extensão
- C# e Java são bastante semelhantes no contexto da OOP, embora haja diferenças na sintaxe e recursos.
- Os genéricos são amplamente utilizados para definir coleções como Listas, Arrays e Dicionários. Tudo isso é restrito por restrições.



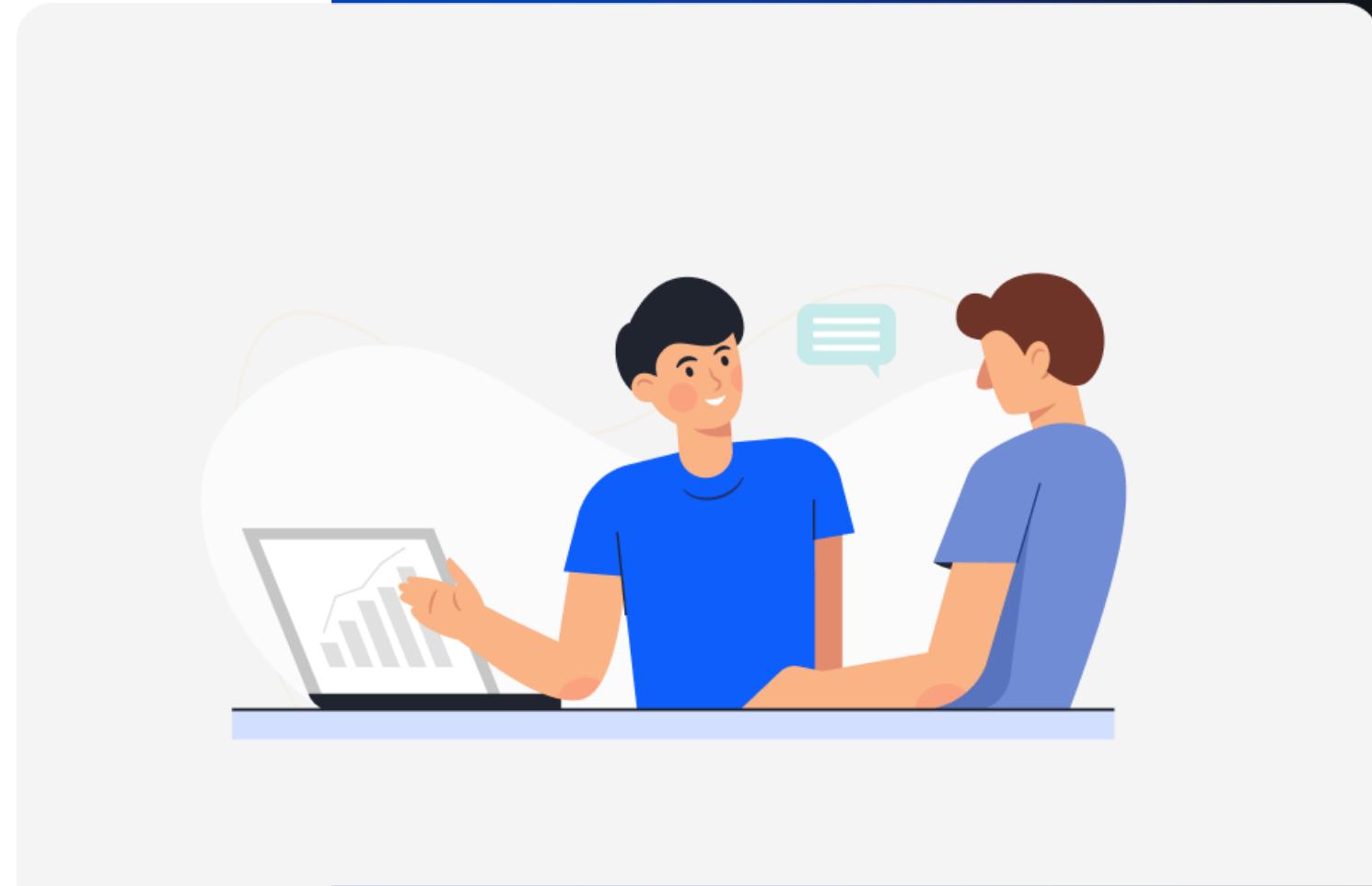
Comentários e Perguntas



Entre em
contato comigo
ou com seu
tutor se precisar
de algum
esclarecimento



Reúna-se com
seus tutores
esta semana
para mais
prática e
conclua sua
tarefa de casa



Obrigado por sua atenção

