Programação C#

Conceitos Iniciais

Profº Ruben Prado

FEVEREIRO - 2025

Sumário

AMBIENTES DE DESENVOLVIMENTO: TIPOS, INSTALAÇÃO E CONFIGURAÇÃO.

O ambiente de desenvolvimento, como o próprio nome diz, é um programa, também conhecido pela sigla, em inglês IDE (Integrated Development Environment), traduzido Ambiente de desenvolvimento integrado. Quando se utiliza um software como esse, temos à nossa disposição diversas ferramentas de edição de código, compilação, testes e distribuição.

Neste universo das IDEs temos muitas opções, desde pagas e gratuitas. No caso específico para o desenvolvimento de jogos com Unity. Utilizaremos o Visual Studio Community. Esta é uma versão gratuita e possui todas as ferramentas necessárias para iniciar o desenvolvimento de um jogo.

Como exemplos de outras IDEs para desenvolvimento C#, além do Visual Studio, temos:

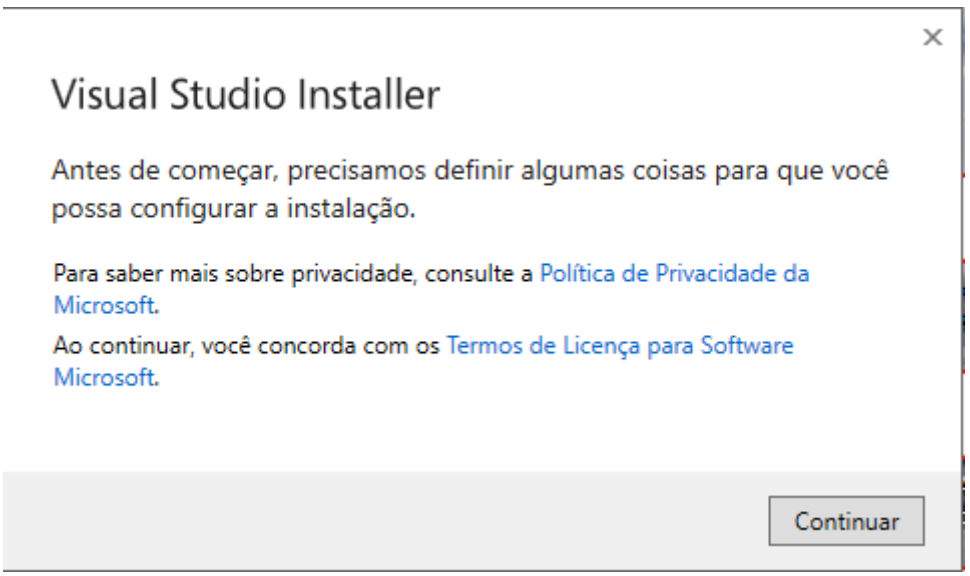
- Jetbrains Rider;
- Visual Studio Code;

Vale uma observação especial, no caso do Visual Studio Code (VScode). Embora possa ser classificado como IDE por alguns, na verdade ele é um editor de código. O grande diferencial desta ferramenta é a disponibilização de uma loja de extensões que permitem funcionalidades extras, tornando este editor em uma IDE de fato tão poderosa quanto as IDEs anteriores.

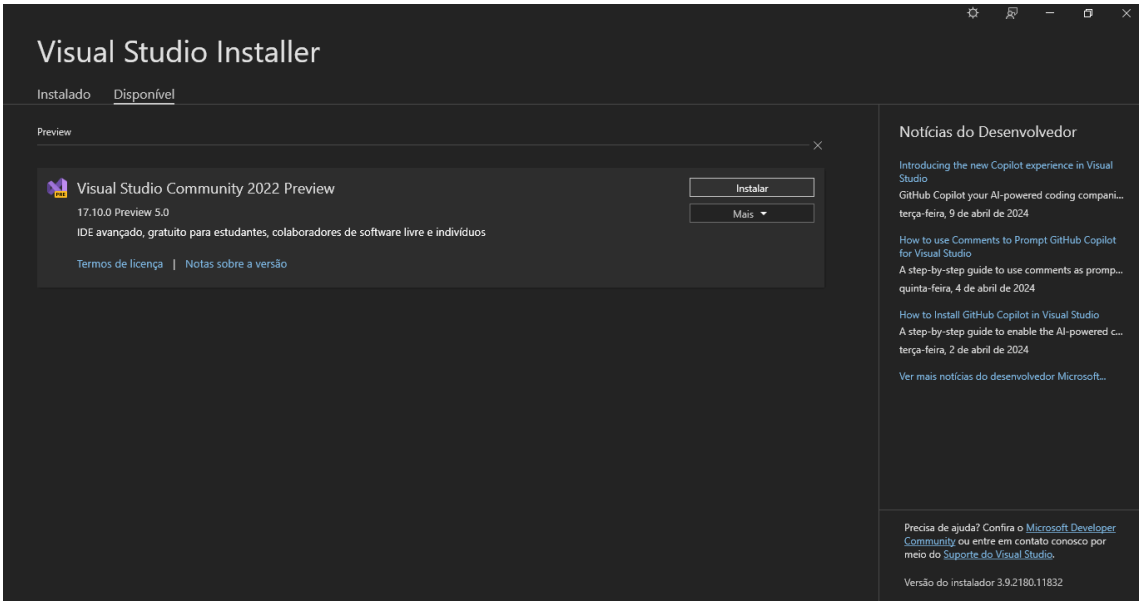
Em uma avaliação de prós e contras o Visual Studio seria a melhor opção por já vir de maneira completa e com as ferramentas necessárias, permitindo que o desenvolvedor poupe tempo na configuração do ambiente.

Pesquise por Visual Studio, ou digite visualstudio.microsoft.com. Independente de quando este site será consultado, procure pela opção de baixar o Visual Studio.

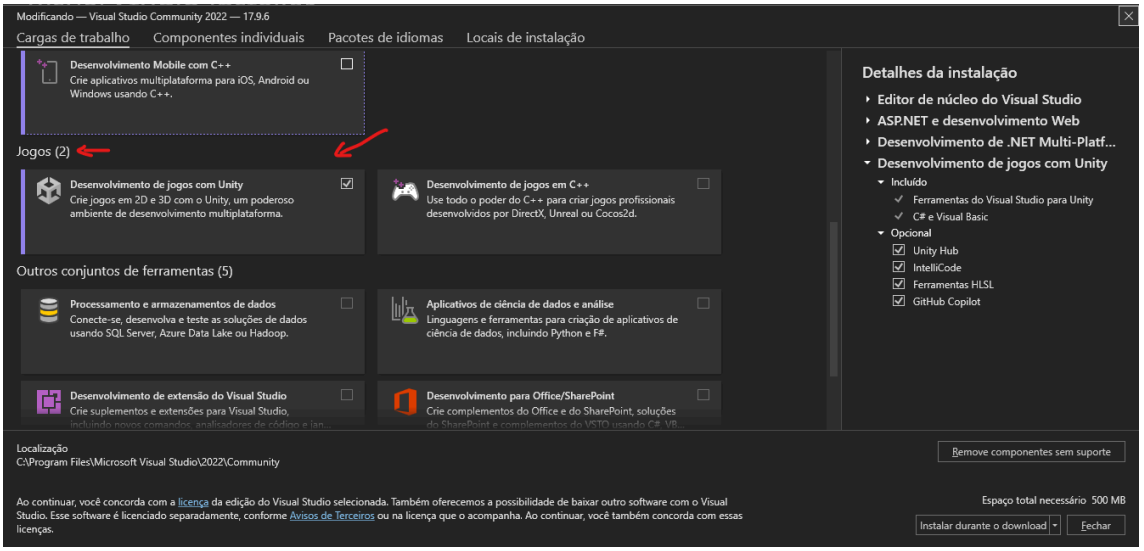
De acordo com a imagem acima, existem duas opções. A IDE completa Visual Studio e o editor Visual Studio Code. Considere avaliar os requisitos de hardware para instalação. O VScode apresenta requisitos menos exigentes, entretanto, as extensões adicionais podem torná-lo tão pesado quando o Visual Studio.



Após realizar o download do arquivo, clique duas vezes para iniciar a instalação. A primeira tela que surge, exige uma verificação do seu sistema. Clique em continuar. Após baixar e instalar o instalador será aberta uma nova janela com a opção de instalação

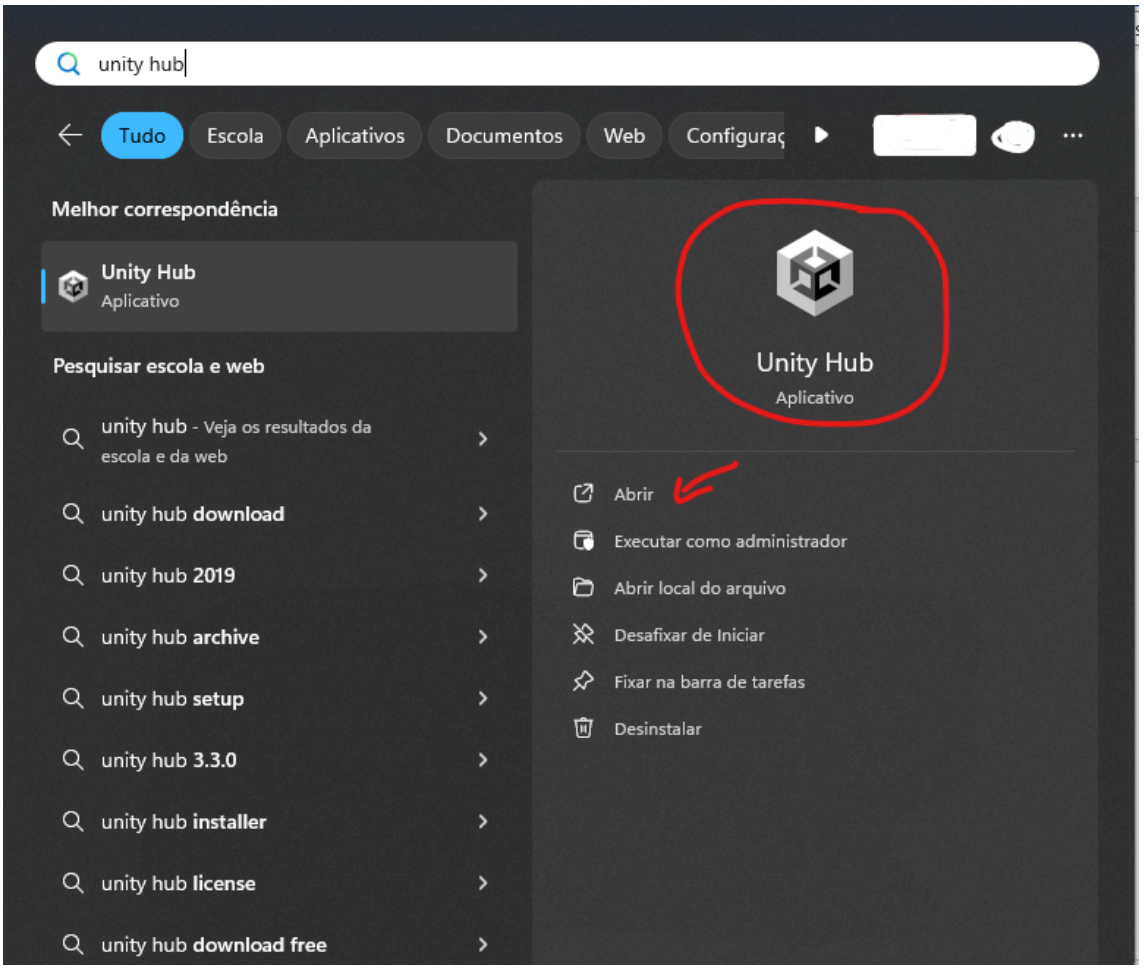


Clique em instalar. Fique atento às opções de instalação. O Visual Studio é um ambiente que possui várias configurações de ambiente de desenvolvimento. Procure pelo grupo Jogos.

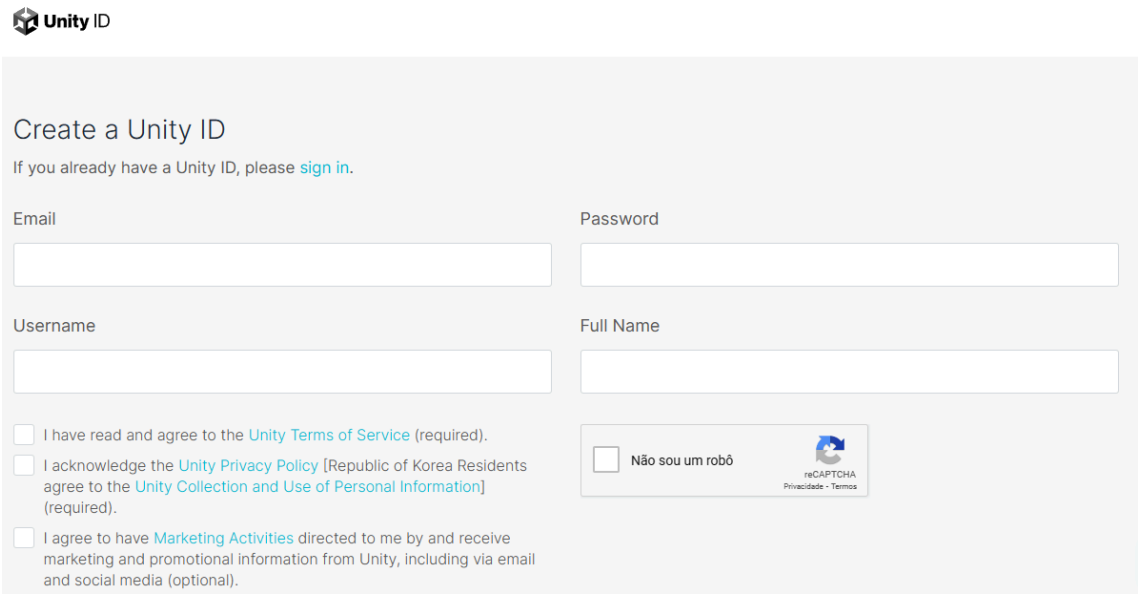


Aqui temos as opções para Desenvolvimento de jogos com Unity e Desenvolvimento de Jogos com C++, que utiliza outras Game Engines, como a Unreal e Coco. Marque a opção da Unity. Em seguida clique na opção de Instalar durante o Download, tornando a instalação mais rápida.

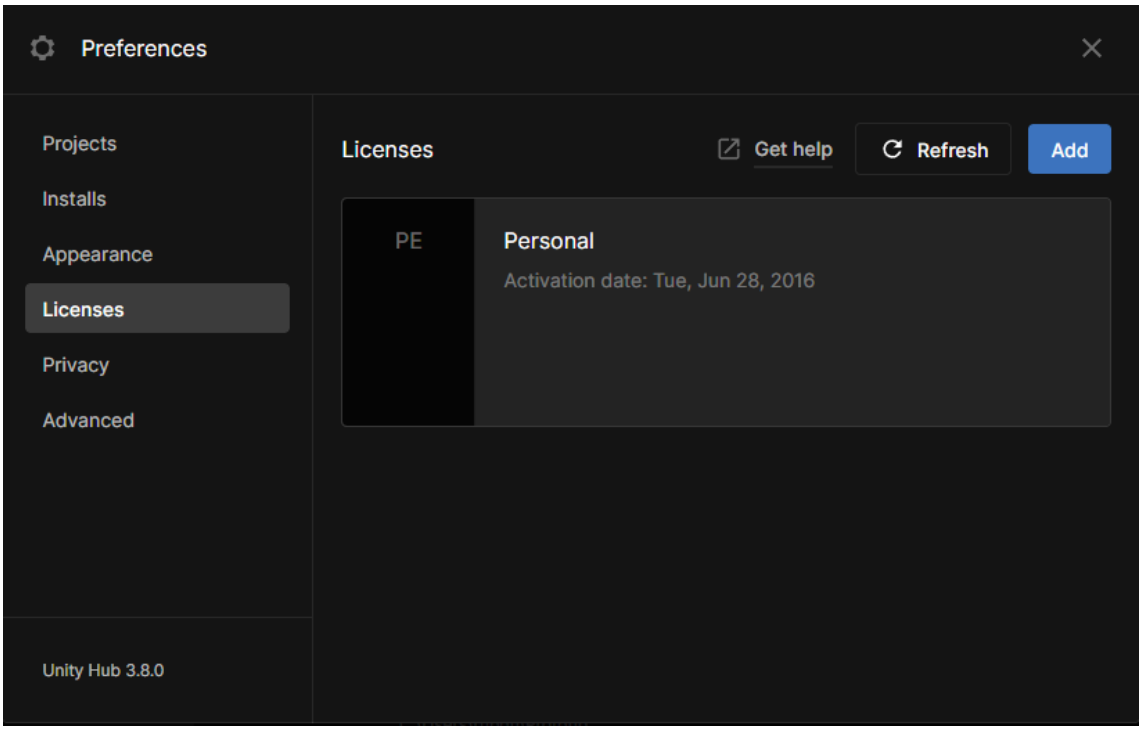
Ao final da instalação você pode abrir o Visual Studio, mas agora precisamos partir para a instalação da Unity. Para isso, procure no Menu Iniciar por, Unity Hub.



Ao iniciar o Unity Hub será necessário a criação de ID Unity. Portanto, utilize a opção para criação de uma conta. Caso já tenha uma, entre com seu usuário e senha. Para criar a conta você será redirecionado para o navegador.



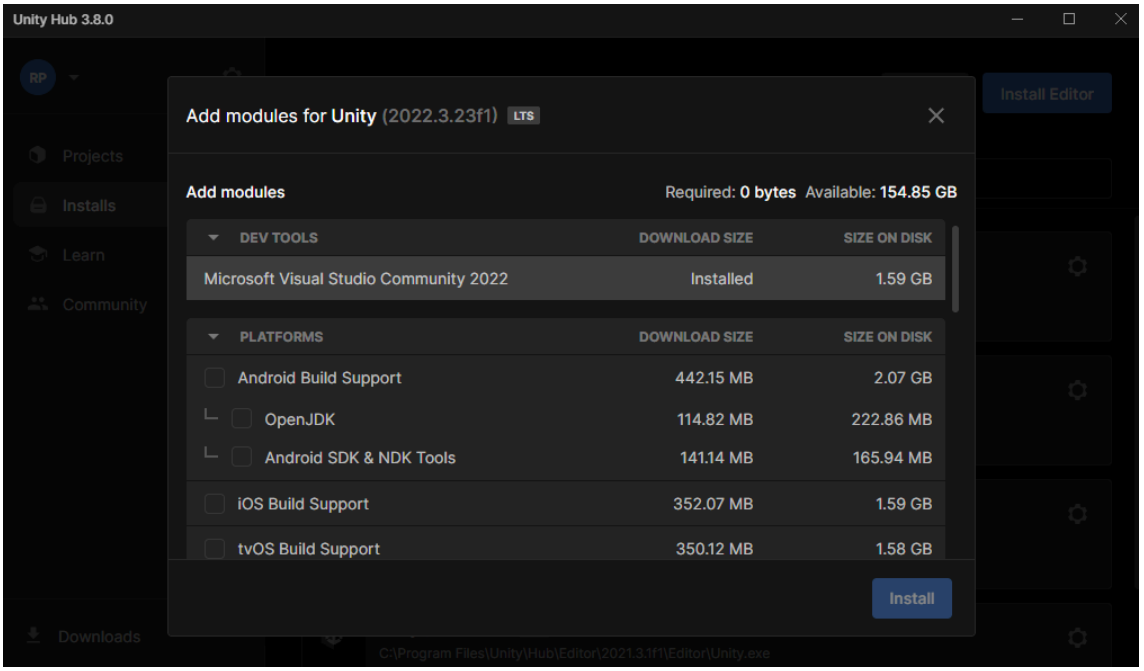
Após a etapa de entrada, ou seja, o processo de login, você será apresentado a uma tela sobre o licenciamento da Unity. Escolha utilizar a licença pessoal (Personal). Escolha: Get a Free personal license, traduzindo Pegue uma licença pessoal gratuita.



A licença de utilização da Unity lhe garante a utilização do ambiente de desenvolvimento e todos os recursos disponíveis, chegando até a publicação

do jogo. Um aviso dado pelo programa diz respeito aos rendimentos, ocasionalmente, adquiridos através da venda de produtos, desenvolvidos com a Unity, que diz: “Se sua receita oriunda de projetos, em conjunto com o uso do Unity, for inferior a US$ 100 mil (ou se a receita e o financiamento agregados de sua empresa forem inferiores a US$ 100 mil) nos últimos 12 meses, você estará qualificado a usar o Unity Personal”. Mesmo que seu objetivo seja vender jogos, você ainda tem chance de se manter dentro dessa margem de ganhos para permanecer isento.

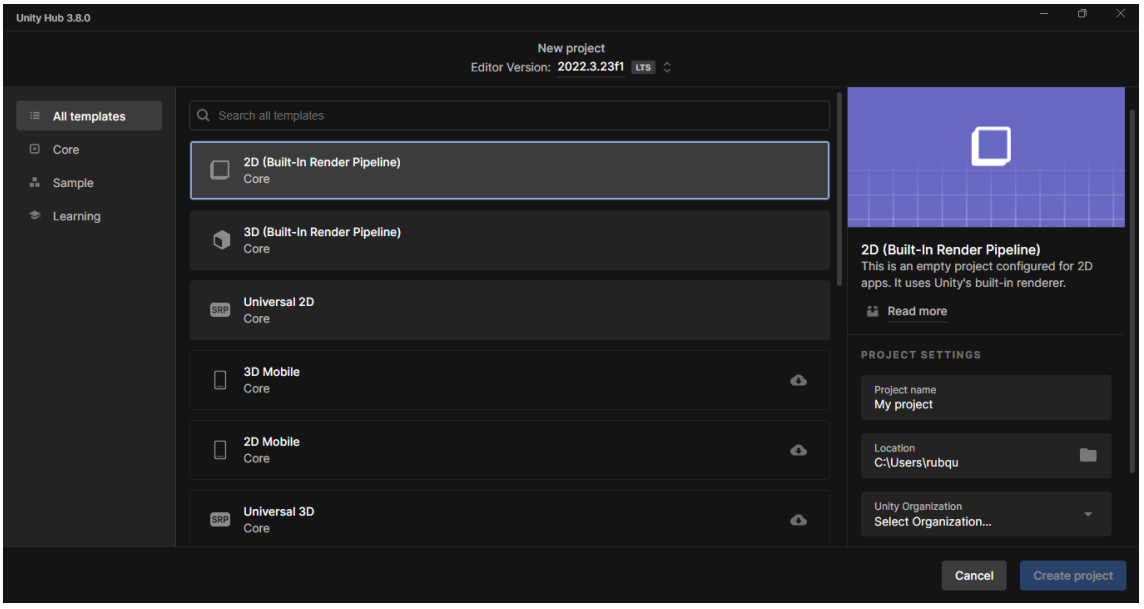
Após a aquisição da licença será necessário instalar o editor. Sempre é sugerido a última versão LTS (Long-term support), algo que traduzido seja, suporte de longo prazo, significando uma versão estável e testada, contendo menos erros e bugs, e que possui suporte técnico em caso de problema, pela fabricante.



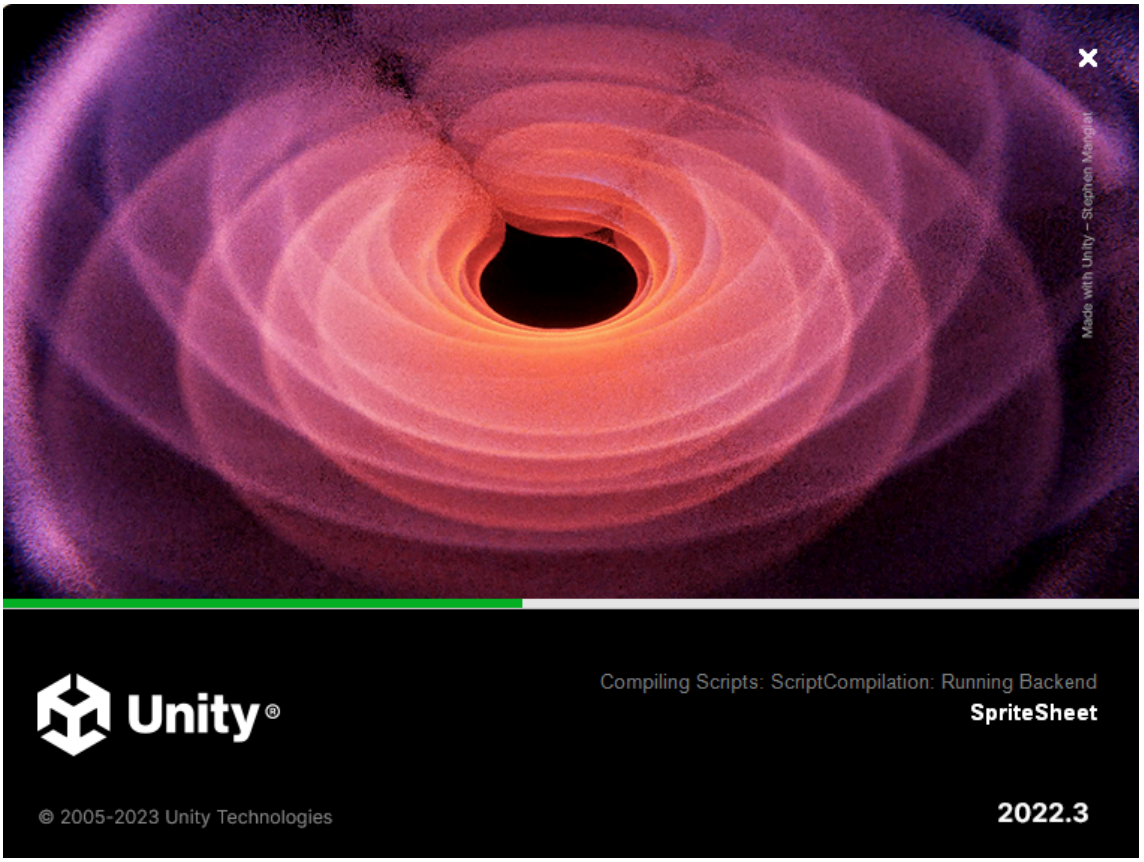
No momento da instalação, será possível a escolha de módulos para desenvolvimento e entrega dos jogos. Observe que a primeira opção é exatamente o Microsoft Visual Studio Community, ou seja, mais uma opção para instalação do ambiente de desenvolvimento é iniciarmos pelo Unity Hub. Outras opções também são as plataformas onde o jogo irá rodar.

Por padrão temos Windows, Mac e Linux. Podemos acrescentar WebGL, Android, IOS, PS4, PS5, Universal Windows Plataform (Xbox) entre outros. Nesse primeiro momento não há necessidade de se preocupar com essas opções, sendo a instalação padrão satisfatória. Há ainda outras opções de linguagem e documentação, mas seguem a mesma lógica. Clique em Install e aguarde a finalização da instalação.

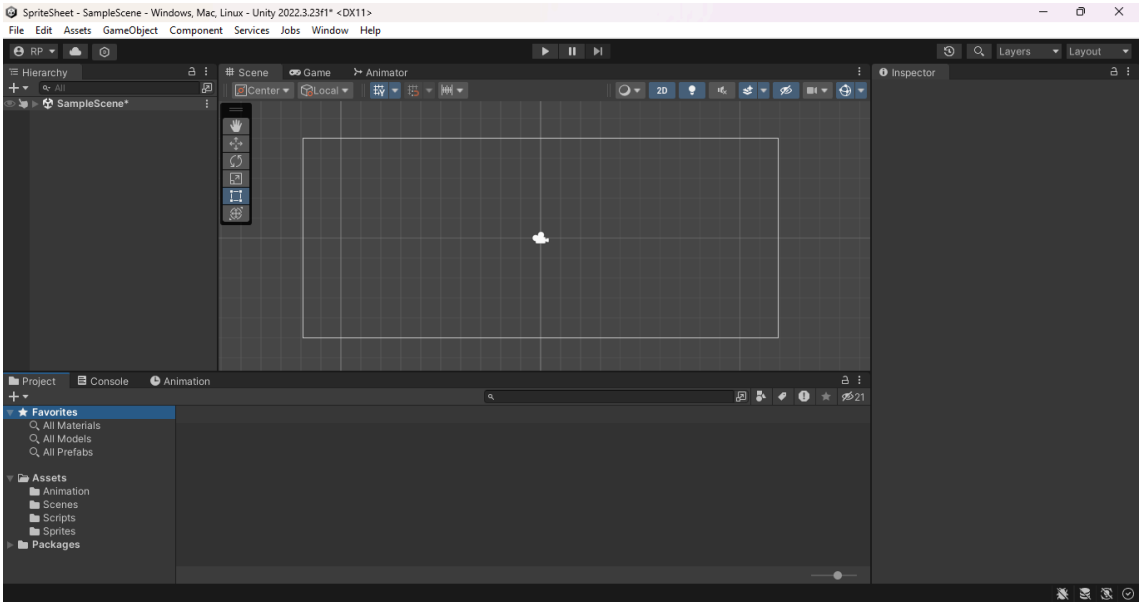
Após a finalização da instalação do editor, podemos finalmente criar um projeto na Unity. Embora estejamos ainda tratando da instalação do ambiente de desenvolvimento, nesta etapa de criação do projeto, estamos finalizando a configuração deste ambiente.



Clique no botão New Project, em seguida, na tela que se segue, temos a configuração inicial de nome e local, também temos, na parte central, a escolha de templates de projetos. Isso significa que podemos pré-configurar os nossos projetos visando qual plataforma o jogo será executado e os tipos de gráficos. Nesse primeiro momento iremos trabalhar com o projeto 2D (Built-In Render Pipeline) Core. Escolha essa opção, nomeie seu projeto com o nome desejado e em seguida clique em Create Project.



Sempre que um novo projeto é criado, o tempo de carregamento do editor pode demorar alguns minutos, dependendo do perfil de hardware para o desenvolvimento. Aguarde até que o editor abrirá.



Ambientes de desenvolvimento, teste, homologação, produção

A instalação desse ambiente será amplamente utilizada para o desenvolvimento de jogos, porém, para um processo de aprendizado mais eficiente, podemos iniciar utilizando apenas o Visual Studio, ou VScode. O trabalho, primeiramente, com a linguagem C# traz uma série de vantagens, desde a aplicação da lógica, até a utilização das estruturas na linguagem. Tudo isso sem distrações da Unity.

Aprender a programar em C# direto na Unity exige um acréscimo de conteúdo a ser compreendido ao mesmo tempo. A Unity disponibiliza uma série de recursos que tornam o desenvolvimento de jogos mais fácil, porém podemos reservar para uma segunda etapa deste aprendizado.

Frameworks, bibliotecas, linguagens, compiladores

Dentro deste universo da programação, alguns conceitos são importantes para compreender a enorme quantidade de opções e recursos disponíveis. Entre elas estão os frameworks, bibliotecas, linguagens e compiladores.

A seguir seguem de forma clara e objetiva 4 conceitos importantes. São eles:

- Framework: está relacionada a uma estrutura abstrata que serve de molde para o desenvolvimento, organização e publicação de software de forma geral. Um framework pode ser focado em uma linguagem específica, mas é acompanhado de um conjunto de bibliotecas que auxiliam na programação. A Unity, pensando nela como uma Game Engine, pode ser considerada um framework, ou seja, ela é constituída de uma série de facilidades que aceleram e facilita o desenvolvimento de jogos, como por exemplo o sistema de física, partículas, interface, inputs, materiais etc. Imagine que sem uma Game Engine precisaríamos criar vários desses sistemas a partir do zero, assim como nos primórdios da programação de jogos, assim como o.Net que é um framework mais complexa, que podemos chamar também de plataforma de desenvolvimento.

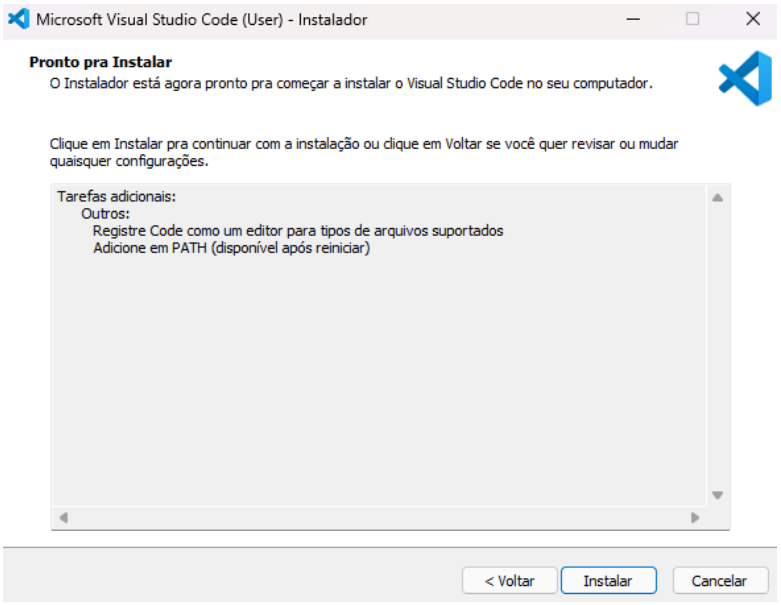
- Biblioteca: este é um conceito interessante, que podemos inicialmente, compreender exatamente como o nome sugere. Biblioteca é um conjunto de livros, onde podemos ter diversos temas distintos. Muito conhecimento que nos auxilia a ter uma compreensão mais ampla da vida e de nossas atribuições como pessoa. Passado esse conceito mais analítico/filosófico, passemos agora para o conceito prático relativo à programação. A biblioteca é um conjunto de códigos que podem ser implementados em um projeto para auxiliar no desenvolvimento de aplicações. Podemos dizer que são códigos que contém cálculos e algoritmos que podem ser reutilizados e adaptados aos nossos projetos. Um framework, grosso modo, pode ser visto como um conjunto de bibliotecas. Como por exemplo a biblioteca System do .Net, que provê métodos que utilizamos para receber e exibir dados na tela.
- Linguagem: Ao desenvolver um programa, jogo, aplicativo, ou seja, qualquer software, precisamos definir qual a linguagem será utilizada para abstrairmos a lógica dos algoritmos criada pelo programador. Assim como o português e o inglês, para você ser compreendido ao se comunicar com outras pessoas, a linguagem de programação, da mesma forma possui estruturas semânticas, sintáticas, símbolos que permitem comunicar ao computador como a ordem, ações e dados que serão inseridos serão tratados e comunicados de volta. Existem diversas linguagens de programação que podem ser utilizadas para diversas finalidades. O .Net, por exemplo, permite a utilização não só do C#, mas de outras linguagens, entretanto a Unity, não aceita essas outras linguagens, sendo desenvolvida e disponibilizada, apenas para a linguagem C#.
- Compilador: É um software que analisa e executa o código escrito em uma linguagem de programação. Ele pode ser considerado o intérprete entre o programador e a máquina. Ele acusa os erros que impedem a execução do código gerando uma validação antes de compilar, ou seja, transformar o código de alto nível em código de baixo nível. Código de máquina, bits, zeros e uns. Todo o código é reduzido a esse nível para que a máquina possa processar as solicitações do código.

Configuração e início dos trabalhos;

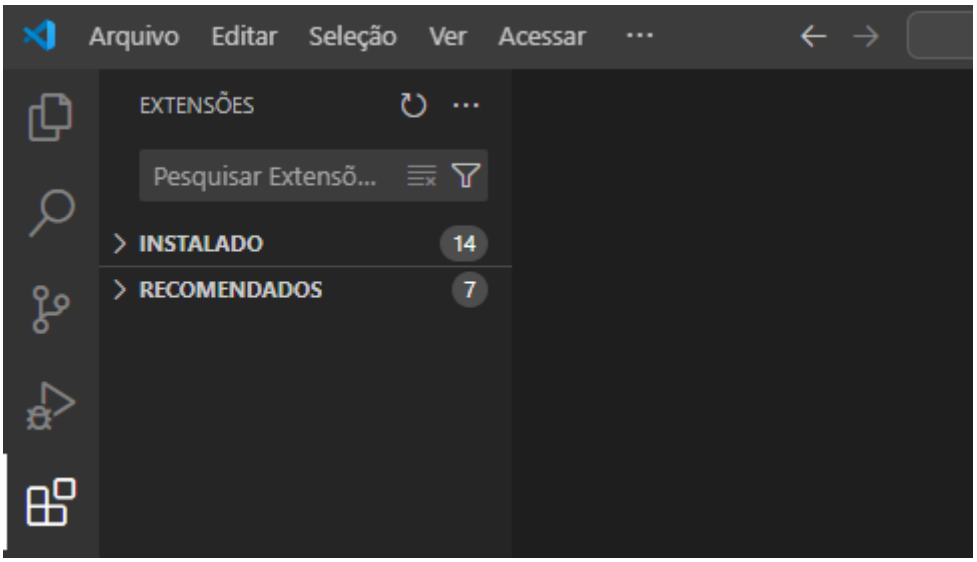
No início deste capítulo realizamos a instalação e configuração iniciais do ambiente de desenvolvimento de jogos utilizando a Unity. Mas iremos realizar também o aprendizado da linguagem C#, como etapa primordial deste plano. Para tanto iremos instalar e configurar o VScode para o aprendizado da linguagem C# e posteriormente, utilizaremos o Visual Studio para programação de jogos com a Unity.

De volta ao site, para download do Visual Studio Code, clique em Baixe o Visual Studio Code, escolha a versão do seu sistema operacional.

Você será redirecionado para o endereço code.visualstudio.com, com início de um download automático. Nesta mesma página há uma série de indicações de recursos e materiais de instrução para melhorar a experiência de utilização do VScode. Após o download inicie a instalação do programa.

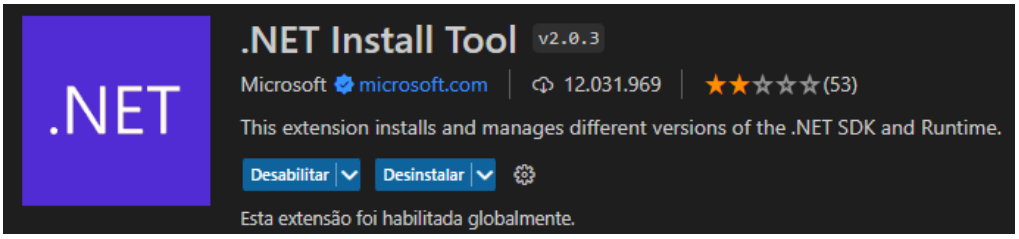


Na instalação clique em próximo, mantenha as opções padrão e para finalizar clique em Instalar. Após finalizar inicie o VScode. Após abrir o programa clique em Extensões.

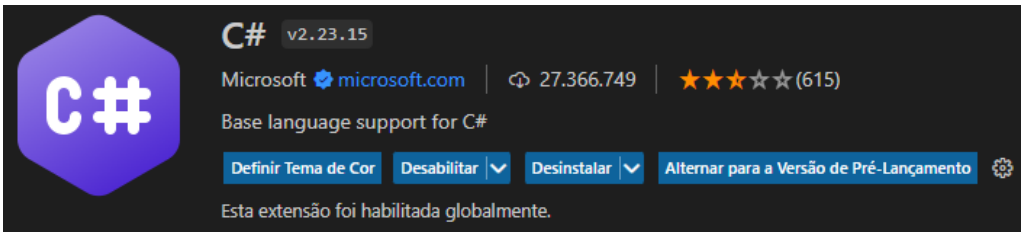


No campo Pesquisar Extensões, procure pelas seguintes extensões:

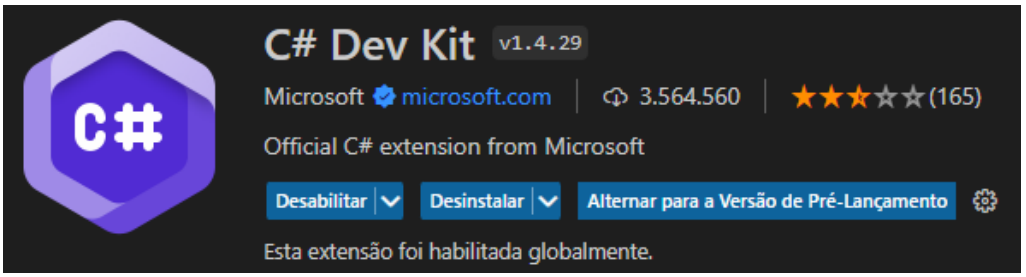
- .Net Install Tool, da Microsoft;



- C#, da Microsoft;



- C# Dev Kit, da Microsoft;



- IntelliCode for C# Dev Kit, da Microsoft;

- Portuguese (Brazil) ... (Opicional)

Caso seja necessário, será solicitado a reinicialização do programa.

LÓGICA DE PROGRAMAÇÃO: CONCEITOS E APLICABILIDADES.

Revisão dos conceitos de Lógica

Quando queremos criar ou desenvolver um software para realizar determinado tipo de processamento de dados, devemos escrever um programa, devemos escrevê-lo usando uma linguagem que tanto o computador quanto o criador de software entendam. Essa linguagem é chamada de linguagem de programação (Ascencio e Campos)

As etapas para desenvolvimento de um programa são:

- Análise – Estudo do problema a ser solucionado e as informações a serem processadas;
- Algoritmo – Solução em linguagem visual ou textual; e
- Codificação – Passagem do Algoritmo para a linguagem escolhida.

“Algoritmo é a descrição de uma sequência de passos que deve ser seguida para a realização de uma tarefa.” (ASCENCIO, 1999)

ALGORITMO – FAZER UM SANDUÍCHE

- Pegar o pão
- Cortar o pão ao meio
- Pegar a maionese
- Passar a maionese no pão
- Pegar e cortar alface e tomate
- Colocar alface e tomate no pão
- Pegar o hambúrguer
- Fritar o hambúrguer
- Colocar o hambúrguer no pão.

ALGORITMO – MULTIPLICAR 2 NÚMEROS

- Receber os dois números que serão multiplicados
- Multiplicar os números
- Mostrar o resultado obtido na multiplicação

Atividade

Você tem um lobo, um carneiro e uma cesta de repolho, e precisa levar todos eles para o outro lado do rio. Porém, o seu barco só pode levar um de cada vez. Mas, se você deixar o lobo e o carneiro sozinhos, o lobo come o carneiro. Se deixar o carneiro e a cesta de repolho, o carneiro come a cesta de repolho. Como você os levará até o outro lado do rio? Crie um algoritmo para resolver este problema.

Variável

Em um programa são recebidos dados que precisam ser armazenados para serem utilizados no processamento. O armazenamento é feito na memória.
Os computadores trabalham com o sistema numérico binário (0 e 1). Os dados são convertidos em bits e bytes (tabela ASCII) para serem armazenados na memória e acessados por meio de um endereço.
A variável representa essa posição na memória. Ela possui nome e tipo de dado. O seu conteúdo pode variar ao longo do tempo de execução do programa.

AS VARIÁVEIS NÃO PODEM SER IDENTIFICADAS COMEÇANDO COM NÚMEROS, POSSUIR ESPAÇOS e CARACTERES ESPECIAIS ou PALAVRAS RESERVADAS

Tipos de Dados

- Numéricos – podem ser números inteiros ou reais.
- Inteiros – positivos ou negativos, sem ponto flutuante.
- Reais – positivos ou negativos, com ponto flutuante.
- Lógicos – Conhecidos como booleanos, são verdadeiro ou falso.
- Caracteres – Um único caractere ou uma cadeia de caracteres. São textos com letras maiusculas e minúsculas, além de números e caracteres especiais.

Operadores

Operador	Exemplo	Descrição
=	X = Y	Operador de atribuição. O conteúdo de Y é passado para X.
+	X + Y	Soma do valor de X e de Y.
-	X – Y	Subtração do valor de Y pelo de X.
*	X * Y	Multiplica o valor de X pelo de Y.
/	X / Y	Divide o valor de X pelo de Y.
%	X % Y	Retorna o valor do resto da divisão de X por Y.
++	X++	Incrementa o valor de X em 1.
--	X--	Decrementa o valor de X em 1.
==	X == Y	Valor de X é igual ao de Y.
!=	X != Y	Valor de X é diferente de Y.
<=	X <= Y	Valor de X menor ou igual a Y.
>=	X >= Y	Valor de X maior ou igual a Y.
<	X < Y	Valor de X menor que Y.
>	X > Y	Valor de X maior que Y.

Estrutura Condicional

A estrutura condicional permite que um determinado comando só seja executado caso a condição de uma sentença seja verdadeira. Uma condição é uma comparação que possui dois valores possíveis: verdadeiro ou falso.

```
SE condição
ENTÃO INÍCIO
```

```
        comando1
        comando2
        FIM
```

A estrutura condicional também pode ser composta, entretanto seu funcionamento continua sendo o mesmo. Se a primeira condição for falsa o segundo bloco de comandos será executado

```
SE condição
ENTÃO   INÍCIO
        comando1
        comando2
        FIM
SENÃO   INÍCIO
        comando3
        comando4
        FIM
```

Operadores Lógicos

Os operadores são utilizados nas condições para permitirem conjunção, disjunção e negação respectivamente E, OU e NÃO.

E	OU	NÃO
V e V = V	V ou V = V	NÃO V = F
V e F = F	V ou F = V	NÃO F = V
F e V = F	F ou V = V	
F e F = F	F ou F = F	

A estrutura CASE é uma estrutura condicional. Quando há muitas opções exclusivas, podemos utilizar essa estrutura para não correr o risco das outras serem executadas.

```
ESCOLHA <expressão-de-seleção>
CASO  <exp1>
      <sequência-de-comandos-1>
CASO  <exp2>
      <sequência-de-comandos-2>
```

```
OUTROCASO
    <sequência-de-comandos-extra>
FIMESCOLHA
```

Estruturas de Repetição

Uma estrutura de repetição é utilizada quando um trecho do algoritmo ou até mesmo o algoritmo inteiro precisa ser repetido. O número de repetições pode ser fixo ou estar atrelado a uma condição. Assim, existem estruturas para tais situações.

A estrutura “para” é utilizada quando se sabe o número de vezes que um trecho de algoritmo deve ser repetido.

```
para(inteiro i = 0; i < 8; i++)
{
//Codigo a ser executado enquanto a condição for satisfeita.
}
```

A estrutura “enquanto” é utilizada quando não se sabe o número de vezes que um trecho do algoritmo deve ser repetido, embora também possa ser utilizada quando se conhece esse valor. Utiliza-se uma condição para análise e a repetição é feita enquanto essa condição mostra-se verdadeira.

```
logico condicao = verdadeiro
enquanto (condicao)
{
//Executa a as instruções dentro do laço enquanto a condicao for
verdadeira
}
```

A estrutura “repita” ou “Faça enquanto” é utilizada quando não se sabe o número de vezes que um trecho do algoritmo deve ser repetido, embora também possa ser utilizada quando se conhece esse número. Essa estrutura

diferencia-se da estrutura “enquanto” pois as instruções serão executadas pelo menos uma vez, antes da condição ser testada.

```
logico condição = verdadeiro
faca{
/*Executa os comandos pelo menos uma vez,
e continua executando enquanto a condição for verdadeira */
} enquanto(condicao)
```

LINGUAGEM C#

Sintaxe básica C#

`//` Comentário em uma linha.

`/* */` Bloco de comentário.

`( )` Separador de expressões, como cálculos e comparações, também é usado para colocar os parâmetros de comandos e funções.

`{ }` Delimitador de bloco de comandos.

`;` Todos os comandos "simples" terminam com ";" .

Recomendações Microsoft

- O caractere de sublinhado é permitido, mas seu uso não é recomendado;
- Palavras-chave da linguagem C# não podem ser usadas como identificadores; int int;
- Procure iniciar o nome com uma letra minúscula;
- Evite usar todas as letras maiusculas; (use somente para definir constantes)
- Quando o nome tiver mais de uma palavra, a primeira letra de cada palavra após a primeira
- deve ser maiuscula (conhecido como notação camelCase); `int idadeDoFuncionario`
- Use nomes em caixa alta para nomear apenas constantes: `float PI= 3.1416;`
- Use nome significativos que descrevam o objetivo da variável: `decimal valorDoPrecoFinal;`
- Não inicie o nome as variáveis com números ou caracteres especiais:
    - `- string $nomeDoCliente;`
    - `float 1_ValorProduto;`
    - `int #idadeAluno;`
- Não use a notação Húngara: Ex: `sNome, dblValor, nIdade`, etc.
- Nao crie variaveis com o mesmo nome mudando somente entre maiúsculas e minúsculas;

- `meuValor, MeuValor`

Declaração de variáveis

Em C#, uma variável é um espaço de memória identificado por um nome, onde um dado de um tipo específico pode ser armazenado e manipulado ao longo da execução do programa. A declaração de uma variável envolve especificar seu tipo e seu identificador, e, opcionalmente, inicializá-la com um valor.

Estrutura Básica

A sintaxe básica para declarar uma variável em C# é a seguinte:

```
tipo nomeVariavel;
```

Exemplos:

Declarando uma variável do tipo inteiro:

```
int idade;
```

Declarando uma variável do tipo string:

```
string nome;
```

Inicialização de Variáveis

É comum inicializar uma variável no momento de sua declaração, atribuindo um valor inicial:

```
int idade = 30;
string nome = "João";
```

Também é possível declarar múltiplas variáveis do mesmo tipo em uma única linha:

```
int a = 10, b = 20, c = 30;
```

Tipos de Dados

Em C# os tipos de dados podem ser divididos em:

Tipos Numéricos: int, long, float, double, decimal

Tipo Caractere: char

Tipo Booleano: bool
Tipo Texto: string
Tipos Complexos: classes, structs, arrays, etc.

O tipo da variável determina o tamanho, o intervalo de valores que ela pode armazenar e as operações permitidas.

Declaração com Inferência de Tipo
A partir do C# 3.0, o compilador permite a utilização do modificador var para declaração de variáveis com inferência de tipo. Nesse caso, o tipo é determinado automaticamente com base no valor atribuído:

```
var idade = 30; // O compilador inferirá que 'idade' é do tipo int.
var nome = "João"; // O compilador inferirá que 'nome' é do tipo string.
```

Apesar da conveniência, recomenda-se utilizar **`var`** apenas quando o tipo da variável estiver claro para facilitar a manutenção e a legibilidade do código.

Nomes e palavras reservadas

Os identificadores (nomes de variáveis, métodos, classes, etc.) em C# devem seguir certas regras:

- Devem começar com uma letra (a–z, A–Z) ou um sublinhado (_).
- Não podem começar com números.
- Podem conter letras, dígitos e o caractere de sublinhado.
- São sensíveis a maiúsculas e minúsculas (por exemplo, `idade` e `Idade` são considerados diferentes).
- Devem ser significativos e descritivos para facilitar a compreensão do código.

Exemplos de bons identificadores:

```
int idadeFuncionario;
string nomeCompleto;
```

```
double salarioMensal;
```

Exemplos de identificadores inválidos:

- `1valor` (não pode começar com número)
- `nome do cliente` (não pode conter espaços)
- `int` ou `class` (palavras reservadas)

Palavras Reservadas

Palavras reservadas são palavras que têm um significado especial para o compilador e não podem ser utilizadas como identificadores. Essas palavras são parte da sintaxe da linguagem e definem a estrutura do código.

Algumas palavras reservadas comuns em C# incluem:

- **Tipos de dados:** `int`, `string`, `bool`, `double`, `decimal`, etc.
- **Controle de fluxo:** `if`, `else`, `switch`, `case`, `for`, `while`, `foreach`, `break`, `continue`
- **Modificadores de acesso e outros:** `public`, `private`, `protected`, `internal`, `static`, `readonly`, `abstract`, `virtual`, `override`, `sealed`
- **Palavras estruturais:** `class`, `struct`, `interface`, `namespace`, `using`, `return`

Caso seja necessário utilizar uma palavra reservada como identificador, é possível fazê-lo precedendo o nome com o caractere `@`. Por exemplo:

```
string @class = "MinhaClasse"; // 'class' é uma palavra reservada, mas com @ pode ser utilizada.
```

No entanto, o uso do `@` deve ser evitado quando possível, para manter a clareza do código.

Convenções de Nomenclatura

Além das regras sintáticas, é importante seguir as convenções de nomenclatura para manter o código legível e consistente:

- **CamelCase:** Utilizado para nomes de variáveis e parâmetros (ex.: `idadeFuncionario`, `nomeCompleto`).
- **PascalCase:** Utilizado para nomes de classes, métodos e propriedades (ex.: `Funcionario`, `CalcularSalario`, `DataNascimento`).
- **Nomes de Constantes:** Geralmente, são escritos em maiúsculas com separação por underline (ex.: `PI`, `TAXA_MAXIMA`).

Essas convenções não são impostas pelo compilador, mas ajudam a manter a clareza e a consistência do código em projetos colaborativos.

Métodos e Parâmetros

Métodos tem alguns requisitos básicos:

- O tipo de dado a ser retornado;
- Um nome único, iniciando com uma letra maiuscula;
- Um par de parênteses seguindo o nome do método;
- Um par de chaves seguindo o corpo do método (onde as instruções são declaradas).

Seguindo essas regras temos:

```
modificadorDeAcesso tipoRetorno NomeUnico(tipoDeParametro nomeDoParametro)
{
	corpo do método;
}
```

Nomeando:

- Assim como variáveis, os métodos, devem ter nomes únicos, significativos e distintos. Métodos definem ações, portanto é bom ter isso em mente quando nomeá-los.
- Sempre iniciam com letra maiúscula, não possuem espaços e quando o nome é composto, as palavras que se seguem iniciam com a letra maiúscula. Ex.: GeradorDePersonagem()

Definindo Parâmetros:
Quando declaramos um método podemos definir parâmetros adicionais que podem ser passados para sua execução;
Para isso precisamos definir:

- Tipo explícito; e
- Nome único;

Como uma variável, mas apenas localmente e definida apenas para o recebimento dos dados necessários à execução do método.

Definindo Argumentos:

- Se definir Parâmetros são as variáveis definidas para que o método receba dados adicionais, esses dados são os Argumentos;
- Eles precisam ser do tipo definido pelo parâmetro do método;
- Podem ser apenas valores ou uma variável declarada em outro local do código;
- Nomes de Argumentos e Parâmetros não precisam combinar!

Definindo Retorno:

- Os métodos do tipo “void”, não tem retorno, ou seja, executam um bloco de código e encerram, embora os códigos executados pelo método tenham seus resultados.
- Para entender melhor o retorno. Pense nele como uma resposta, na qual você recebe, ou não quando solicita que uma pessoa faça uma determinada tarefa para você. Alguns tipos de tarefas você provavelmente não receberá nenhum feedback, mas até poderá, ou não, ver suas consequências. Em outros casos, existem tarefas que você precisa de retorno. Às vezes uma mensagem afirmativa, ou negativa, às vezes um valor relativo a algum pagamento feito, um troco, de forma

geral alguma notícia. Portanto, ao desenvolver determinados métodos, alguns retornos podem ser necessários.

Definindo Retornos:

Já os métodos que precisam de retorno. Ele precisa ser especificado pelo tipo. Conforme exemplo a seguir:

```
public int GenerateCharacter(string name, int level)
{
 Debug.LogFormat("Character: {0} - Level: {1}", name, level);
 return level += 5;
}
```

O tipo de retorno (Int, Float, String, etc.)

A palavra-chave RETURN fecha o método. O que se segue será o retorno. O valor do tipo Inteiro. O que estiver depois dela não é executado.

Operadores

Operadores aritméticos

- + (Adição)
- - (Subtração)
- * (Multiplicação)
- / (Divisão)
- % (Resto/Módulo)

Operadores de atribuição

- = Atribuição simples
- += Atribuição aditiva
- -= Atribuição Subtrativa
- *= Atribuição Multiplicativa
- /= Atribuição de divisão
- %= Atribuição de módulo

Operadores Relacionais

- == Igualdade

- > Maior
- < Menor
- <= Menor igual
- >= Maior igual
- != Diferente

Operadores Lógicos

- && AND
- || OR
- ! NOT

Estrutura de decisão

- Estrutura do Se-Então (If-Else) em linguagem C#
- Utilização dos Operadores Lógicos;
- If-Else aninhado;
- Switch;

Uma das funções mais básicas de um computador é controlar o que acontece quando ele se depara com situações pré-determinadas. Em um jogo, quando você clica, pressiona uma tecla, seu personagem atinge é atingido por um objeto, atinge um determinado nível resultados são esperados para essas ações. Quando codificamos não é diferente. Escrevemos o código prevendo os resultados que serão esperados às ações previstas no design do jogo. Determinamos condições que irão mudar ou não, influenciar ou não, um determinado comportamento dentro do programa ou do jogo. Estamos criando um fluxo, como um caminho. Prevemos alternativas para situações diferentes, caso elas aconteçam.

Controle de Fluxo com If Else

```
if(condicao verdadeira)
{
      bloco de código;
```

```
}
else
{
      bloco de código;
}
```

A estrutura ELSE é opcional e só é adicionada caso seja necessário. Ela será acionada no caso da expressão declarada no if não for verdadeira.

```
if(condicao verdadeira)
{
      bloco de código;
}
else if(condicao verdadeira)
{
      bloco de código;
}
else{
      bloco de código;
}
```

No caso de ser necessário mais de um teste, outro IF pode ser adicionado junto do ELSE, criando um bloco ELSE IF. Uma nova expressão é declarada para ser testada. Se for verdadeira, será executada. O bloco ELSE continua sendo opcional. Se nenhum dos dois IFs for executado ele será executado.

```
if(vida < 50)
{
      Debug.Log(“Vida Pela Metade”);
}
else if(vida <= 25 )
{
      Debug.Log(“Regenere sua Vida”);
}
else{
      Debug.Log(“Vida Ok”);
```

```
}
```

Operadores de comparação podem ser utilizados nas expressões. Uma nova expressão é declarada para ser testada. Se for verdadeira, será executada.

Operador NOT

Se quiser que a expressão teste uma condição falsa, há duas opções.

```
if(condicao == falso)
{
      bloco de código;
}
```

Especificando o operador de comparação e definindo o valor booleano para falso.

```
if(!condicao)
{
      bloco de código;
}
```

Outra opção é utilizar o operador lógico NOT, invertendo o valor verdadeiro para falso.

```
if(!condicao == “valor”)
{
      bloco de código;
}
```

Pode “excluir” um determinado valor.

Utilização dos Operadores Lógicos

Quando existem múltiplas condições a serem testadas é possível combinar em uma única expressão utilizando os operadores lógicos.

```
if(armaEquipada && tipoArma == “Espada”)
{
      bloco de código;
}
```

&& equivale ao **E**, quando as duas condições precisam ser verdadeiras.

```
If(tipoArma == “Espada” || tipoArma == “Martelo”)
{
      bloco de código;
}
```

|| equivale ao **OU**, quando pelo menos uma das condições precisa ser verdadeira.

IF Aninhado

```
if(armaEquipada)
 {
        Um novo bloco IF é chamado, caso o anterior seja verdadeiro.
        if(tipoArma == “espadaLonga")
        {
            Debug.Log("Avante!");
        }
 }
else
 {
      Se arma estiver equipada e a arma for uma espada longa, então diga: Avante! Senão Diga:
      Debug.Log("Punhos não são eficientes contra armaduras...");
 }
```

Switch

Quando se tem mais de três condições a serem testadas, pode ser uma boa alternativa utilizar a estrutura Switch Case.

```
switch(expressao) Expressão a ser testada para se gerar um valor.
{
      O valor será verificado com as alternativas configuradas em cada CASE.
```

```
  case valor1:
  bloco de Código;
  break;
  case valor2:
  bloco de Código;
  break;
  default: O DEFAULT é executado caso nenhum dos CASEs seja chamado!
  bloco de Código;
  break; Os comandos BREAK servem para parar a execução do SWITCH após uma das opções ser executada.
}
```

Estrutura de repetição.

Estruturas de repetição:

- For;
- While;
- Foreach;

Estruturas de Repetição:

- Estruturas de repetição são uma forma de iteração por uma coleção de dados;
- Pode ser utilizado para realizar uma tarefa repetidas vezes;
- Verificar uma condição por repetidas vezes;

FOR

FOR: São utilizados para repetir um bloco de código um número de vezes antes de seguir em frente a execução do programa.

```
for(inicializador; condição; iterador)
{
      bloco de código;
}
```

```
List<string> QuestPartyMembers = new List<string>()
{ "Grim the Barbarian", "Merlin the Wise"};
```

```
for (int i = 0; i < QuestPartyMembers.Count; i++)
{
 Debug.LogFormat("Index: {0} - {1}", i, QuestPartyMembers[i]);
}
```

WHILE

WHILE: Podem ser similares às estruturas IF no quesito de rodar caso a expressão seja verdadeira.

```
Inicializador
while (condição)
{
 bloco de código;
 iterador;
}
```

```
int PlayerLives = 3;

 while(PlayerLives > 0)
 {
    Debug.Log("Still alive!");
    PlayerLives--;
 }
```

FOR EACH

FOR EACH: Cada elemento de uma coleção é armazenado em uma variável local, sendo acessível para utilização. O tipo da variável deve ser o mesmo tipo do elemento da coleção.

```
foreach(tipoDoElemento nomeLocal in colecaoVariavel)
{     bloco de código;  }
```

```
List<string> QuestPartyMembers = new List<string>()

{ "Grim the Barbarian", "Merlin the Wise", "Sterling the Knight"};
```

```
foreach(string partyMember in QuestPartyMembers)
{
 Debug.LogFormat("{0} - Here!", partyMember);
}
```

Cuidados com repetição infinita

É necessário tomar cuidado com as iterações infinitas, ou loops infinitos.

Existem expressões que são colocadas que impossibilitam a parada do loop em uma estrutura de repetição.

Quando isso acontece o programa trava, ou até mesmo fecha. É necessário verificar se as condições, os operadores estão corretamente definidos, maior, menor. Se um determinada condição está incrementando ou decrementando conforme o esperado.

ARQUITETURA DE ALGORITMOS: CONCEITOS E APLICABILIDADES.

Este tópico apresenta os fundamentos e as diversas aplicações dos algoritmos, abordando desde a definição e as características essenciais que os compõem até as suas aplicações em áreas como o desenvolvimento de games e a inteligência artificial. Compreender a arquitetura dos algoritmos é fundamental para a construção de soluções eficientes e para a implementação de lógicas avançadas em C#.

Conceito de Algoritmo

Um algoritmo pode ser definido como uma sequência finita e ordenada de instruções, bem definidas e executáveis, destinadas à resolução de um problema específico. Independentemente da linguagem utilizada, o algoritmo descreve a lógica de solução de um problema, servindo de base para a implementação de programas de computador.

Os algoritmos podem ser representados por meio de:

- Pseudocódigo: Uma forma de descrever a lógica de maneira estruturada e próxima da linguagem natural.
- Fluxogramas: Diagramas que ilustram, de forma visual, o fluxo lógico das operações.
-

Exemplo Prático em C#

A seguir, um exemplo simples de algoritmo implementado em C#, que realiza a soma de dois números:

```
// Algoritmo para somar dois números
Console.Write("Digite o primeiro número: ");
int num1 = Convert.ToInt32(Console.ReadLine());
Console.Write("Digite o segundo número: ");
int num2 = Convert.ToInt32(Console.ReadLine());
int soma = num1 + num2;
Console.WriteLine("A soma é: " + soma);
```

Além disso, é importante introduzir conceitos básicos de análise de algoritmos, como a complexidade de tempo e a complexidade de espaço, que medem o desempenho e a eficiência de um algoritmo.

Algoritmo e Games

No desenvolvimento de jogos, os algoritmos desempenham um papel crucial ao definir a lógica que governa o comportamento dos elementos do jogo e a dinâmica das interações. Dentre as aplicações mais comuns, destacam-se:

- Game Loop: O ciclo principal do jogo que atualiza o estado, processa entradas e renderiza os gráficos. Este loop precisa ser otimizado para garantir uma experiência de jogo fluida.
- Detecção de Colisões: Algoritmos que verificam se dois ou mais objetos colidem durante a execução do jogo, essenciais para a interação entre personagens e o ambiente.
- Pathfinding: Técnicas como o algoritmo A* são amplamente utilizadas para determinar rotas ideais para a movimentação de personagens e inimigos dentro do jogo.
- Geração de Números Aleatórios: Fundamental para introduzir variabilidade, seja no comportamento dos inimigos ou na criação de cenários e níveis.

A linguagem C# é uma das principais escolhas no desenvolvimento de games, especialmente quando associada a engines como o Unity. Um algoritmo bem implementado pode, por exemplo, controlar o comportamento de um personagem inimigo ou gerenciar a movimentação dos objetos na tela.

Inteligência Artificial

A Inteligência Artificial (IA) é o campo da ciência da computação que busca desenvolver sistemas capazes de realizar tarefas que exigem inteligência humana. Em C#, a implementação de algoritmos de IA varia desde simples regras de decisão até complexos modelos de aprendizado de máquina.

Algoritmos Comuns em IA

Busca e Otimização:

Algoritmos de busca, como a Busca em Largura (BFS), Busca em Profundidade (DFS) e A*, são empregados na resolução de problemas e na navegação em ambientes complexos.

Algoritmo Minimax:
Utilizado em jogos competitivos, o algoritmo minimax permite que um agente tome decisões otimizadas, especialmente quando combinado com a técnica de poda alfa-beta para reduzir a quantidade de cálculos.

Redes Neurais e Aprendizado de Máquina:
Embora mais complexos, os modelos de redes neurais podem ser implementados com o auxílio de bibliotecas como o ML.NET, permitindo que aplicações em C# realizem tarefas de reconhecimento de padrões, classificação e previsão.
A integração de IA em projetos com C# pode resultar em sistemas de recomendação, análise de dados e jogos mais desafiadores, nos quais os adversários controlados pelo computador apresentam comportamentos realistas e adaptativos.

ESTRUTURA DE DADOS: CONCEITOS, TIPOS E APLICABILIDADES.

Coleções de Dados:

- Arrays;
- Lists;
- Dictionaries;

Até agora vimos que para armazenar cada tipo de dado utilizamos as variáveis. Elas armazenam apenas um tipo de dado por vez. Entretanto existem situações em que podemos precisar de armazenar um grupo de dados. Para isso temos os tipos de coleções (Collection Types) em C#, que incluem Array, List e Dictionarie. Cada um desses tipos tem as suas vantagens e desvantagens de utilização.

Arrays

O ARRAY é o tipo mais básico de coleção. Podemos pensar nele como um tipo de container que contém um grupo de valores, também chamados de elementos. Esses elementos podem ser acessados e modificados individualmente.

ARRAY:

- Qualquer tipo de valor pode ser armazenado;
- Todos os elementos precisam ser do mesmo tipo;
- O tamanho do Array (Length) ou a quantidade de elemento que ele pode ter, pode ser definido na sua criação e não pode ser modificado depois;
- Elementos podem ser adicionados ou removidos após a sua criação;

Declarando um ARRAY:

```
tipoDoElemento[] nome = new tipoDoElemento[numeroDeElementos];
```

É necessário especificar o tipo de elemento que será inserido no Array, seguido de colchetes. A palavra reservada, NEW, cria um espaço reservado na

memória para armazenar o novo Array. Quantidade de elemento que irão compor o Array.

Exemplo:

```
int[] pontuacoesJogador = new int[3];
```

Inicializando um ARRAY:

Inicialização mais completa

```
int[] pontuacoesJogador = new int[] {713, 589, 973};
```

Inicialização mais curta

```
int[] pontuacoesJogador = {713, 589, 973};
```

ÍNDICE DO ARRAY (INDEX):

- Cada elemento do Array é armazenado na ordem em que ele é adicionado e referenciado a um índice;
- O índice começa em 0 (zero);

Índice de um ARRAY:

```
int[] pontuacoesJogador = new int[] {713, 589, 973};
```

Para passar o valor de um determinado elemento do Array para a variável, basta colocar o nome dele seguido da posição entre os colchetes.

```
int pontuacao = pontuacoesJogador[1];
// O valor de pontuacao será 589
```

Podemos atribuir um novo valor à posição do elemento.

```
pontuacoesJogador[1] = 1587;
```

Listas

Lists (Listas) são similares aos Arrays. Podem ser consideradas mais fáceis de se trabalhar adicionando, removendo e atualizando elementos. O tamanho dos elementos pode ser alterado após a sua declaração.

Declarando uma LIST:

```
List<tipoDoElemento> NomeDaLista = new List<tipoDoElemento>();
```

Usa-se a palavra chave List, declarando o tipo do dado do elemento entre < >.
A palavra reservada, NEW, cria um espaço reservado na memória.
Na List não é declarado o tamanho da lista.

Exemplo:

```
List<string> MembrosDaEquipe = new List<string>();
```

Inicializando uma LIST:

```
List<string> MembrosDaEquipe = new List<string>()
      {
            “Bárbaro Grim”,
            “Mérlim o Sábio”,
            “Cavaleiro Sterling”
      };
```

ACESSANDO E MODIFICANDO LISTs:
Diferentemente do Array, onde utilizamos o índice para realizar as operações de acesso e modificação, na List temos métodos para adicionar, inserir e remover elementos.

Inserindo, Adicionando elementos a LIST:
O método ADD joga o novo elemento para o final da lista.

```
MembrosDaEquipe.Add(“Craven o Necromancer”);
```

Se quisermos inserir em uma posição específica é necessário utilizar o método INSERT, passando a posição e o valor do elemento.

```
MembrosDaEquipe.Insert(1,“Bárbaro Grim”);
```

Removendo elementos da LIST:

Utilizando o método REMOVEAT, remove o elemento que se encontra na posição determinada.

```
MembrosDaEquipe.RemoveAt(0);
```

Utilizando REMOVE e passando como argumento o valor do elemento.

```
MembrosDaEquipe.Remove("Bárbaro Grim");
```

Dicionários

Dictionary armazena pares de valores, chave e valor. Array e List apenas valor; Dicionarios são desordenados. Necessário quando precisa-se armazenar dois valores referenciados;

Declarando um Dictionary:

```
Dictionary<tipoDaChave, TipoDoValor> NomeDoDicionario =
                new Dictionary<tipoDaChave, TipoDoValor>();
```

Usa-se a palavra chave Dictionary, declarando o tipo do dado da chave e do elemento entre < >.

A palavra reservada, NEW, cria um espaço reservado na memória.

Na List não é declarado o tamanho da lista.

Exemplo:

```
Dictionary<string, int>  ItemInventario = new Dictionary<string, int>();
```

Inicializando uma DICTIONARY:

```
Dictionary<string, int> ItemInventario = new Dictionary<string, int>();
      {
            {"Poção", 5}
            {"Antídoto", 7}
            {"Apirina", 1}
      };
```

Inserindo, Adicionando e removendo elementos do DICTIONARY:

O valor de uma chave pode ser acessado como índice, de forma semelhante

ao Array.

```
int numeroDePocoes = ItemInvetario[“Poção”];
```

Caso faça referência a um elemento que não exista, este elemento será criado, com chave e valor.

```
ItemInvetario[“Poção”] = 10;
```

O método ADD joga o novo elemento para o final da lista.

```
ItemInvetario.Add(“Faca de Arremesso”, 3);
```

Utilize o método REMOVE para remover uma elemento do dicionário.

```
ItemInventario.Remove(“Antídoto”);
```

PROGRAMAÇÃO ESTRUTURADA E ORIENTAÇÃO A OBJETO: DIFERENÇAS ENTRE AS ABORDAGENS, CONCEITOS DE ORIENTAÇÃO A OBJETO, CLASSES E OBJETOS.

Este tópico apresenta as diferenças fundamentais entre a programação estruturada e a programação orientada a objetos, detalha os conceitos essenciais da orientação a objeto e discute a utilização de classes, objetos, construtores, encapsulamento, herança, polimorfismo, componentes e interfaces. A compreensão desses conceitos é crucial para a construção de sistemas robustos e modulares, facilitando a manutenção e a escalabilidade dos softwares desenvolvidos em C#.

Programação Estruturada e Orientada a Objetos

A programação estruturada é um paradigma que organiza a solução de problemas por meio da decomposição do programa em funções ou procedimentos. Cada procedimento é responsável por uma tarefa específica, e o programa é estruturado utilizando comandos sequenciais, condicionais e de repetição. Suas principais características são:

- Modularidade: Divisão do programa em blocos ou funções.
- Controle de Fluxo: Uso de estruturas como if/else, loops (for, while) para direcionar a execução.
- Simplicidade: Foco em resolver problemas através de uma sequência linear de instruções.

Programação Orientada a Objetos (POO)

Na programação orientada a objetos, o software é modelado em torno de objetos, que representam entidades do mundo real ou conceitos abstratos. Cada objeto possui atributos (dados) e métodos (comportamentos) e interage com outros objetos por meio de mensagens. As características fundamentais da POO incluem:

- Abstração: Simplificação de entidades complexas, representando apenas suas características essenciais.

- Encapsulamento: Ocultamento dos detalhes internos, expondo apenas a interface necessária para interação.
- Herança: Permite a criação de novas classes baseadas em classes existentes, facilitando a reutilização de código.
- Polimorfismo: Possibilita que objetos de diferentes classes sejam tratados de forma uniforme, através de métodos comuns.
- Comparação:

Estruturada: Foca em funções e procedimentos.
Orientada a Objetos: Foca em objetos e suas interações, promovendo maior modularidade e reutilização.

Classes e Objetos
Classes
Uma classe é um molde que define os atributos e métodos comuns a um grupo de objetos. Em C#, utiliza-se a palavra-chave class para declarar uma classe.

Exemplo:
public class Pessoa

```
{
    // Atributos
    public string Nome;
    public int Idade;

    // Método
    public void Apresentar()
    {
          Console.WriteLine("Olá, meu nome é " + Nome + " e tenho " +
Idade + " anos.");
    }
}
```

Objetos

Um objeto é uma instância concreta de uma classe, possuindo os atributos e comportamentos definidos por ela.

Exemplo:

```
Pessoa p = new Pessoa();
p.Nome = "João";
p.Idade = 30;
p.Apresentar();  // Saída: Olá, meu nome é João e tenho 30 anos.
```

Construtores

Construtores são métodos especiais que inicializam um objeto no momento de sua criação. Em C#, o construtor tem o mesmo nome da classe e não possui um tipo de retorno.

Exemplo:

```
public class Pessoa
{
    public string Nome;
    public int Idade;

    // Construtor
    public Pessoa(string nome, int idade)
    {
        Nome = nome;
        Idade = idade;
    }

    public void Apresentar()
    {
          Console.WriteLine("Olá, meu nome é " + Nome + " e tenho " +
Idade + " anos.");
    }
}
```

Para instanciar um objeto:

```
Pessoa p = new Pessoa("Maria", 25);
p.Apresentar(); // Saída: Olá, meu nome é Maria e tenho 25 anos.
```

Encapsulamento

O encapsulamento é o princípio que consiste em proteger os dados de uma classe, permitindo acesso apenas por meio de métodos e propriedades definidos. Em C#, isso é feito utilizando modificadores de acesso como private, public e protected.

Exemplo:

```
public class ContaBancaria
{
    // Campo privado
    private double saldo;

    // Propriedade para acesso controlado
    public double Saldo
    {
        get { return saldo; }
        private set { saldo = value; }
    }

    public ContaBancaria(double saldoInicial)
    {
        Saldo = saldoInicial;
    }

    public void Depositar(double valor)
    {
        if (valor > 0)
            Saldo += valor;
    }
```

```
    public bool Sacar(double valor)
    {
        if (valor <= Saldo)
        {
            Saldo -= valor;
            return true;
        }
        return false;
    }
}
```

Herança

A herança permite que uma nova classe (derivada) herde os atributos e métodos de uma classe existente (base), facilitando a reutilização e a extensão do código.

Exemplo:

```
// Classe base
public class Animal
{
    public string Nome;

    public void Comer()
    {
        Console.WriteLine(Nome + " está comendo.");
    }
}
```

// Classe derivada

```
public class Cachorro : Animal
{
    public void Latir()
    {
        Console.WriteLine(Nome + " está latindo.");
    }
```

```
}
```

Uso da herança:

```
Cachorro dog = new Cachorro();
dog.Nome = "Rex";
dog.Comer();  // Saída: Rex está comendo.
dog.Latir();  // Saída: Rex está latindo.
```

Polimorfismo

O polimorfismo permite que métodos com o mesmo nome se comportem de maneira diferente em classes derivadas. Em C#, utiliza-se os modificadores virtual e override para implementar o polimorfismo.

Exemplo:

```
public class Animal
{
    public string Nome;

    // Método virtual
    public virtual void FazerSom()
    {
        Console.WriteLine("O animal faz um som.");
    }
}

public class Gato : Animal
{
    // Sobrescrita do método virtual
    public override void FazerSom()
    {
        Console.WriteLine(Nome + " miou.");
    }
}

public class Cachorro : Animal
{
    // Sobrescrita do método virtual
    public override void FazerSom()
```

```
    {
        Console.WriteLine(Nome + " latiu.");
    }
}
```

Uso do polimorfismo:

```
Animal animal1 = new Gato { Nome = "Mimi" };
Animal animal2 = new Cachorro { Nome = "Rex" };
animal1.FazerSom(); // Saída: Mimi miou.
animal2.FazerSom(); // Saída: Rex latiu.
```

Componentes

Componentes são unidades modulares de software que encapsulam funcionalidades específicas e podem ser reutilizadas em diferentes projetos. Em C#, componentes podem ser implementados como classes, bibliotecas ou controles, promovendo a criação de sistemas com alta coesão e baixo acoplamento.

Exemplo de Aplicação:

Uma biblioteca de componentes gráficos ou de acesso a dados pode ser utilizada em diversos projetos, garantindo padronização e eficiência.

Interfaces

Interfaces definem contratos que as classes devem seguir, especificando um conjunto de métodos e propriedades sem fornecer sua implementação. Em C#, as interfaces são declaradas com a palavra-chave interface.

Exemplo:

```
public interface IOperacao
{
    void Executar();
}

public class Soma : IOperacao
{
```

```
    public int A { get; set; }
    public int B { get; set; }

    public void Executar()
    {
        Console.WriteLine("A soma é: " + (A + B));
    }
}
```

Uso da interface:

```
IOperacao operacao = new Soma { A = 5, B = 3 };
operacao.Executar();  // Saída: A soma é: 8
```

Qualificadores de Acesso

Os **qualificadores** em C# são palavras-chave que alteram o comportamento dos membros (variáveis, métodos, classes, etc.) e definem características específicas que auxiliam na organização, manutenção e execução do código.

Static

O qualificativo static indica que um membro pertence à própria classe e não a uma instância específica dela. Dessa forma, membros estáticos são compartilhados entre todas as instâncias e podem ser acessados diretamente pelo nome da classe.

Exemplo:

```
public class Utilidades
{
    public static int Somar(int a, int b)
    {
        return a + b;
    }
}
```

// Uso sem instanciar a classe

```
int resultado = Utilidades.Somar(3, 5);
```

Const

A palavra-chave const é utilizada para declarar constantes, ou seja, valores que não podem ser alterados e devem ser conhecidos em tempo de compilação. Geralmente, as constantes são definidas com nomes em letras maiúsculas para facilitar a identificação.

Exemplo:

```
public class Configuracao
{
    public const double PI = 3.14159;
}
```

// Uso da constante

```
double area = Configuracao.PI * Math.Pow(raio, 2);
```

Readonly

O qualificativo readonly permite que um campo seja atribuído apenas no momento da declaração ou no construtor da classe, garantindo que seu valor não seja modificado posteriormente. Ao contrário do const, o valor do readonly pode ser determinado em tempo de execução.

Exemplo:

```
public class Data
{
```

```
    public readonly DateTime DataCriacao;
    public Data()
    {
        DataCriacao = DateTime.Now;
    }
}
```

Abstract

O modificador abstract pode ser aplicado a classes e métodos. Uma classe abstrata não pode ser instanciada e serve como base para outras classes. Métodos abstratos não possuem implementação e devem ser obrigatoriamente implementados nas classes derivadas.

Exemplo:

```
public abstract class Animal
{
    public abstract void EmitirSom();
}

public class Cachorro : Animal
{
    public override void EmitirSom()
    {
        Console.WriteLine("Au Au");
    }
}
```

Virtual e Override

Os qualificadores virtual e override são utilizados para implementar o polimorfismo em C#. Um método declarado como virtual na classe base pode ser sobrescrito (override) em uma classe derivada, permitindo que cada classe forneça uma implementação específica.

Exemplo:

```
public class Veiculo
{
    public virtual void Mover()
    {
        Console.WriteLine("O veículo está se movendo.");
    }
}

public class Carro : Veiculo
{
    public override void Mover()
    {
        Console.WriteLine("O carro está acelerando.");
    }
}
```

Sealed

O modificador sealed é utilizado para impedir que uma classe seja herdada ou que um método virtual seja sobrescrito por classes derivadas. Ele garante a integridade da implementação quando não se deseja permitir extensões.

Exemplo:

```
public sealed class Utilitario
{
    // Métodos e propriedades que não poderão ser alterados por
herança.
}
```

Extern

A palavra-chave extern indica que a implementação de um método é fornecida externamente, geralmente por meio de bibliotecas não gerenciadas. É

comumente utilizada para integração com código nativo (como DLLs em C/C++).

Exemplo:

```
public class Nativo
{
    [System.Runtime.InteropServices.DllImport("user32.dll")]
    public static extern int MessageBox(IntPtr hWnd, string text,
string caption, uint type);
}
```

Partial

O modificador partial permite que a definição de uma classe, struct ou interface seja dividida em múltiplos arquivos. Essa abordagem facilita a manutenção de grandes bases de código, permitindo que diferentes desenvolvedores trabalhem em partes distintas da mesma classe.

Exemplo:

```
Arquivo PessoaParte1.cs:
public partial class Pessoa
{
    public string Nome { get; set; }
}

Arquivo PessoaParte2.cs:
public partial class Pessoa
{
    public int Idade { get; set; }
}
```

Os qualificadores em C# são ferramentas fundamentais para definir comportamentos específicos e melhorar a organização do código. A correta utilização de static, const, readonly, abstract, virtual/override, sealed, extern e

partial contribui para o desenvolvimento de sistemas mais seguros, eficientes e fáceis de manter.

Modificadores de Acesso

Os modificadores de acesso em C# são fundamentais para controlar a visibilidade e a acessibilidade dos membros de uma classe. Eles definem quais partes de um programa podem acessar determinado membro (variável, método, classe, etc.), contribuindo para a segurança, encapsulamento e integridade dos dados.

Em C#, os modificadores de acesso determinam o escopo de visibilidade dos membros. Essa abordagem permite que os desenvolvedores exponham apenas as partes necessárias de uma classe, ocultando detalhes internos e protegendo a integridade dos dados. Entre os modificadores de acesso mais comuns, destacam-se public, private, protected, internal, protected internal e, a partir do C# 7.2, private protected.

Public

O modificador public torna o membro acessível de qualquer parte do código, seja dentro do mesmo assembly ou em assemblies externos. É utilizado para membros que devem ser expostos livremente.

Exemplo:

```
public class Produto
{
    public string Nome;
    public double Preco;
}
```

Neste exemplo, tanto a propriedade Nome quanto Preco podem ser acessadas por qualquer classe que instancie a classe Produto.

Private

O modificador private restringe o acesso ao membro apenas à própria classe na qual ele foi declarado. Essa abordagem é utilizada para encapsular e proteger os dados internos, evitando acesso indevido ou alterações não autorizadas.

Exemplo:

```
public class ContaBancaria
{
    private double saldo;

    public void Depositar(double valor)
    {
        if (valor > 0)
            saldo += valor;
    }

    public double ConsultarSaldo()
    {
        return saldo;
    }
}
```

Neste caso, o campo saldo é privado e só pode ser acessado ou modificado por meio dos métodos públicos.

Protected

O modificador protected permite que o membro seja acessível somente dentro da classe onde foi declarado e nas classes derivadas. Esse modificador é útil para criar uma hierarquia de classes onde membros internos possam ser utilizados pelas subclasses, sem expor esses detalhes para o mundo exterior.

Exemplo:

```
public class Veiculo
{
    protected string Modelo;
}

public class Carro : Veiculo
{
    public void DefinirModelo(string modelo)
    {
        Modelo = modelo;
    }
}
```

Aqui, o campo Modelo é protegido e pode ser utilizado pela classe Carro, que herda de Veiculo.

Internal

O modificador internal torna o membro acessível apenas dentro do mesmo assembly (projeto). É ideal para componentes que não precisam ser expostos publicamente fora do projeto, mas devem estar disponíveis para todas as classes internas.

Exemplo:

```
internal class GerenciadorDeCache
{
    public void LimparCache()
    {
        // Implementação
    }
}
```

O GerenciadorDeCache pode ser utilizado por qualquer classe dentro do mesmo assembly, mas não será acessível por código externo.

Protected Internal

O modificador protected internal combina as características de protected e internal. Assim, o membro fica acessível tanto para classes derivadas (mesmo em assemblies diferentes) quanto para classes que estejam no mesmo assembly.

Exemplo:

```
public class Base
{
    protected internal int Codigo;
}
```

Dessa forma, Codigo pode ser acessado por classes que herdem de Base e também por classes no mesmo assembly.

Private Protected

Introduzido a partir do C# 7.2, o modificador private protected restringe o acesso ao membro à própria classe e às classes derivadas que estejam no mesmo assembly, combinando uma restrição mais forte do que protected internal.

Exemplo:

```
public class BaseSegura
{
    private protected int Identificador;
}
```

Neste caso, *Identificador* pode ser acessado somente dentro da classe *BaseSegura* e em suas subclasses, desde que estas estejam no mesmo assembly.

Imagine uma classe que representa um sistema de gerenciamento de funcionários. Os dados sensíveis, como o salário, devem ser protegidos:

public class Funcionario

```
{
    public string Nome { get; set; }
    private double salario;

    public Funcionario(string nome, double salario)
    {
        Nome = nome;
        this.salario = salario;
    }

    public double ObterSalario()
    {
        return salario;
    }

    protected void AjustarSalario(double novoSalario)
    {
        if (novoSalario > salario)
            salario = novoSalario;
    }
}
```

Nesse exemplo, Nome é público, enquanto salario é privado e só pode ser acessado por métodos dentro da classe. O método AjustarSalario é protegido, permitindo que subclasses possam modificá-lo de forma controlada.

Os modificadores de acesso são essenciais para garantir o encapsulamento e a segurança do código em C#. Ao definir corretamente os níveis de acesso dos membros, o desenvolvedor pode controlar a visibilidade e proteger a integridade dos dados, promovendo um design mais robusto e modular.

REFERÊNCIAS

DEITEL, Paul J.; DEITEL, Harvey M. **Como Programar em C#**. Pearson, 2013.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. **Introduction to Algorithms**. MIT Press, 2009.

MILLINGTON, Ian. **Artificial Intelligence for Games**. CRC Press, 2019.

BUCKLAND, Mat. **Programming Game AI by Example**. CRC Press, 2005.

NYSTROM, Robert. **Game Programming Patterns**. Genever Benning, 2014.

RUSSELL, Stuart; NORVIG, Peter. **Artificial Intelligence: A Modern Approach.** Pearson, 2020.

Microsoft. **ML.NET Documentation**. Disponível em: https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet

ASCENCIO, Ana Fernanda Gomes. **Fundamentos da Programação de Computadores**. Pearson, 2013.

LIPOVSKY, Stephen. **Programação Orientada a Objetos com C#**. Novatec, 2014.

YOSHIKAWA, Joe. **Component-Based Software Engineering: Putting the Pieces Together**. Addison-Wesley, 2000.