Introdução

- C# (pronuncia-se "C Sharp") é uma linguagem de programação moderna, orientada a objetos e com segurança de tipos.
- C# permite que os desenvolvedores criem muitos tipos de aplicativos seguros e robustos que são executados em .NET.
- C# tem suas raízes na família de linguagens C e será imediatamente familiar para programadores de C, C++, Java e JavaScript.

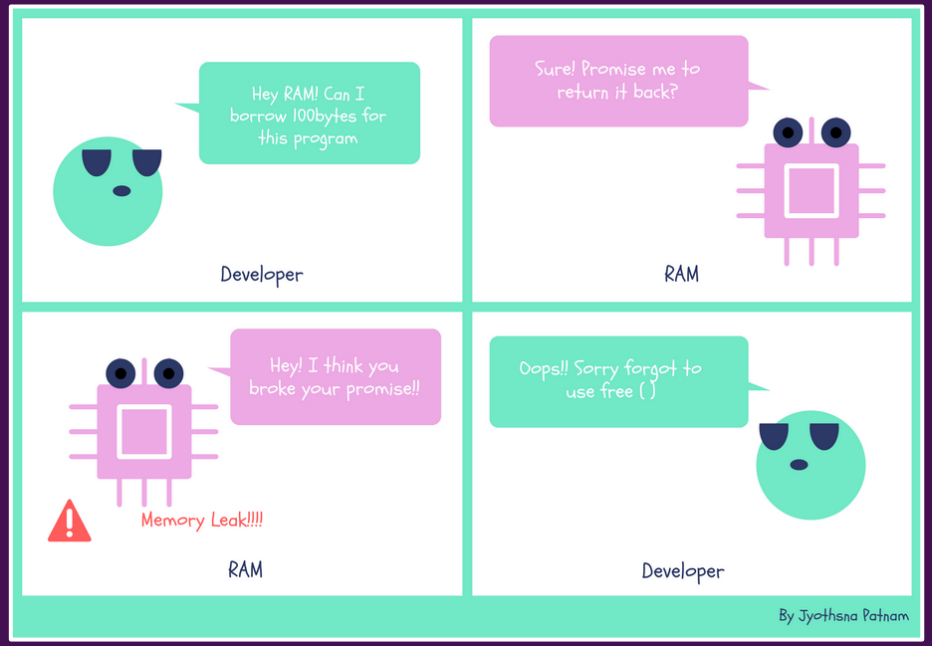
Introdução

- C# é uma linguagem de programação orientada a objetos e componentes.
- C# fornece construções de linguagem para oferecer suporte direto a esses conceitos, tornando o C# uma linguagem natural para criar e usar componentes de software.
- Desde sua origem, o C# adicionou recursos para oferecer suporte a novas cargas de trabalho e práticas emergentes de design de software.
- Em sua essência, C# é uma linguagem orientada a objetos. Você define os tipos e seus comportamentos.

Vários recursos do C # ajudam a criar aplicações robustas e duráveis.

Garbage Collection

- **Garbage collection** recupera automaticamente a memória ocupada por objetos não utilizados inacessíveis.



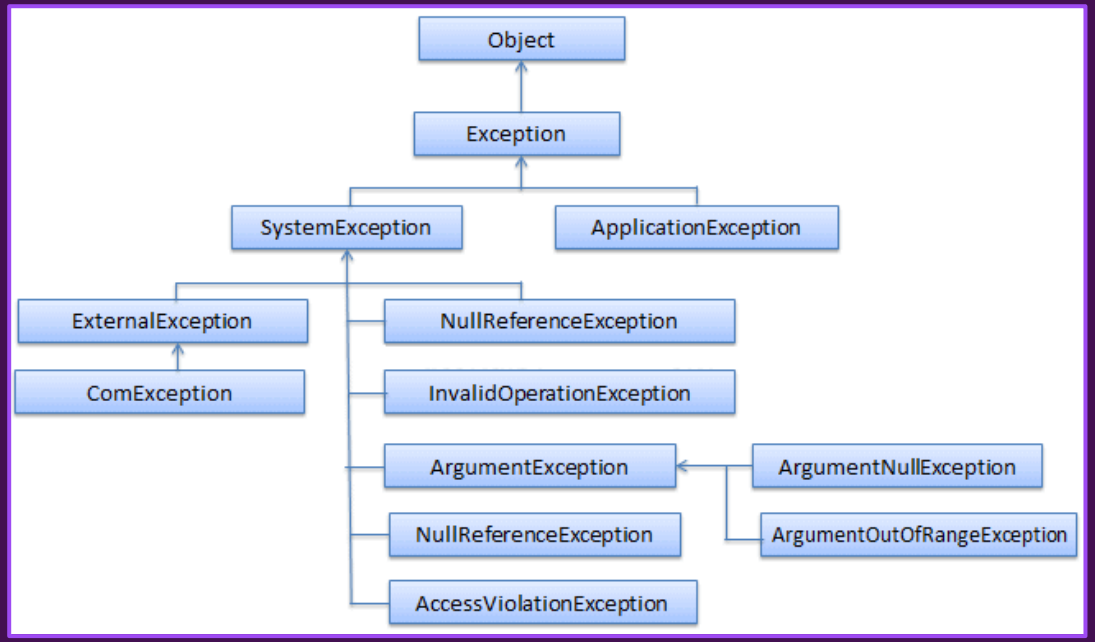
Nullable Types

- **Nullable types** protegem contra variáveis que não se referem a objetos alocados.

```C#
string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

Exception Handling

- **Exception handling** fornece uma abordagem estruturada e extensível para detecção e recuperação de erros



Lambda Expressions

- **Lambda expressions** oferecem suporte a técnicas de programação funcional.

C#

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Language Integrated Query

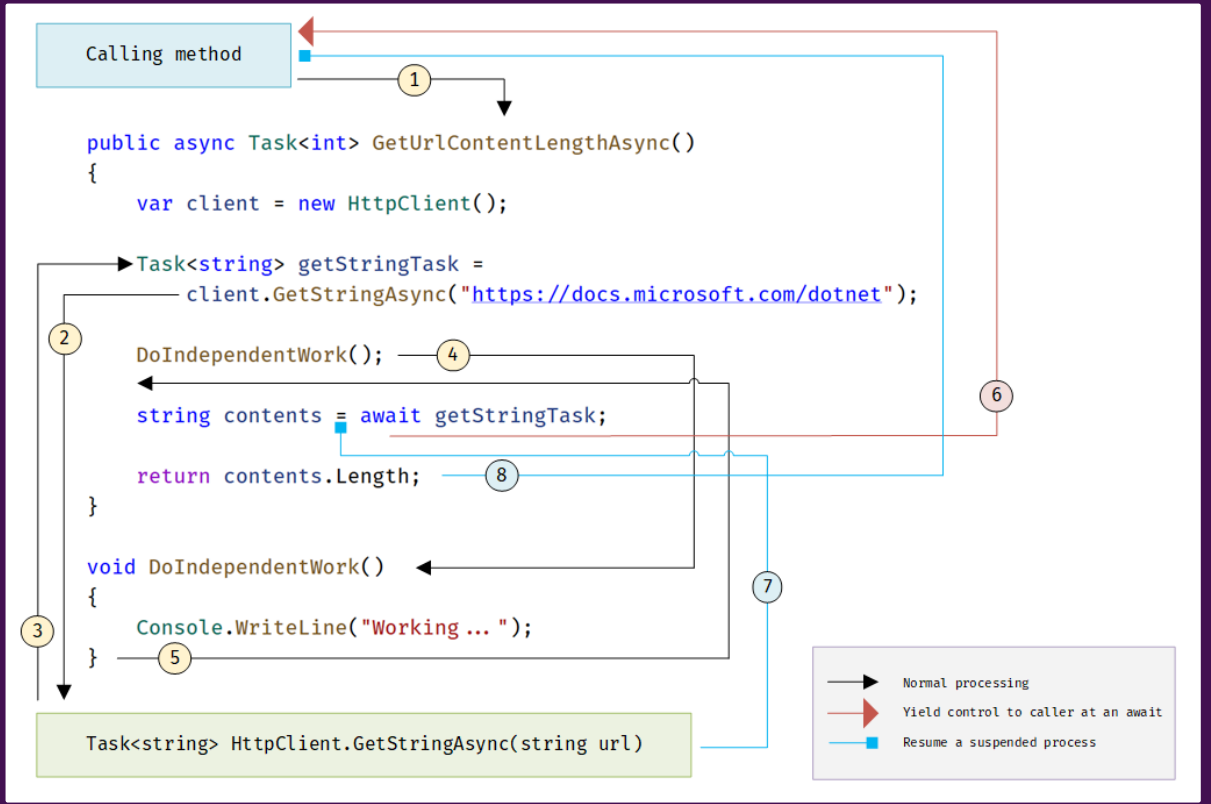
- A sintaxe do **LINQ (Language Integrated Query)** cria um padrão comum para trabalhar com dados de qualquer fonte.

C#

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

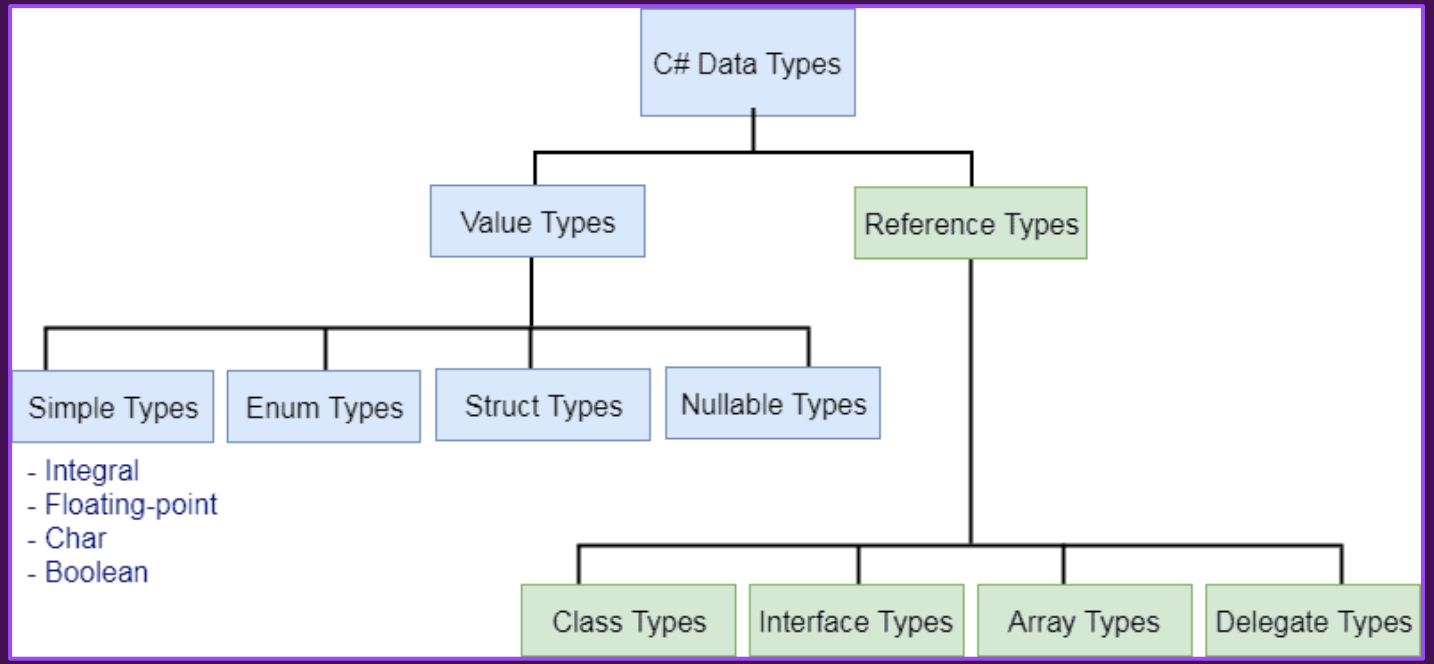
Asynchronous Operations

- O suporte de linguagem para **asynchronous operations** fornece sintaxe para a construção de sistemas distribuídos.



Unified Type System

- C# tem um **unified type system**.



Unified Type System

- Todos os tipos C#, incluindo tipos primitivos, como int e double, herdam de um único tipo raiz object.
- Todos os tipos compartilham um conjunto de operações comuns.
- Valores de qualquer tipo podem ser armazenados, transportados e operados de maneira consistente. Além disso, C# oferece suporte a **reference types** definidos pelo usuário e **value types**.
- C# permite a alocação dinâmica de objetos e armazenamento em linha de estruturas leves.
- C# oferece suporte a métodos e tipos genéricos, que fornecem maior segurança de tipo e desempenho.
- C# fornece iteradores, que permitem aos implementadores de classes de coleção definir comportamentos personalizados para o código do cliente.

Versioning

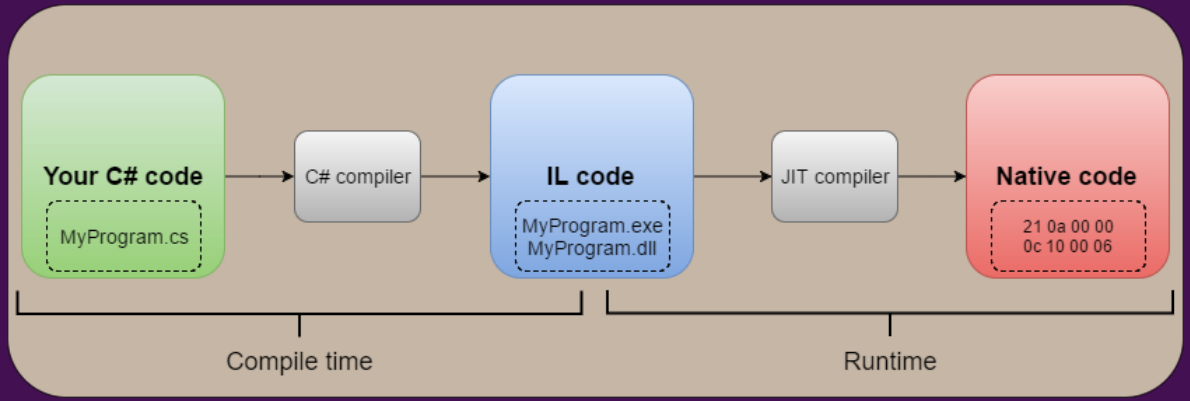
- C# enfatiza o controle de versão para garantir que programas e bibliotecas possam evoluir com o tempo de maneira compatível.
- Os aspectos do design do C# que foram diretamente influenciados por considerações de versão incluem os modificadores virtual e override separados, as regras para resolução de overload de método e suporte para declarações explícitas de membros de interface.

Arquitetura .NET

- Os programas C# são executados em .NET, um sistema de execução virtual chamado **common language runtime** (CLR) e um conjunto de bibliotecas de classes.
- O CLR é a implementação pela Microsoft da **common language infrastructure** (CLI), um padrão internacional.
- A **CLI** é a base para a criação de ambientes de execução e desenvolvimento nos quais linguagens e bibliotecas trabalham juntas de forma integrada.

Arquitetura .NET

- O código-fonte escrito em C# é compilado em uma intermediate language (IL) que está em conformidade com a especificação CLI. O código e os recursos IL, como bitmaps e strings, são armazenados em um assembly, normalmente com uma extensão .dll. Um assembly contém um manifesto que fornece informações sobre os tipos, versão e cultura do assembly.



Arquitetura .NET

- Quando o programa C# é executado, o assembly é carregado no CLR.
- O CLR executa a compilação Just-In-Time (JIT) para converter o código IL em instruções de máquina nativas.
- O CLR fornece outros serviços relacionados à automatic garbage collection, tratamento de exceções e gerenciamento de recursos.
- O código executado pelo CLR às vezes é chamado de "código gerenciado", em contraste com o "código não gerenciado", que é compilado em linguagem de máquina nativa voltada para uma plataforma específica.

Arquitetura .NET

- A interoperabilidade de linguagem é um recurso chave do .NET.
- O código IL produzido pelo compilador C# está em conformidade com a Common Type Specification (CTS).
- O código IL gerado em C# pode interagir com o código gerado nas versões .NET do F#, Visual Basic, C ++ ou qualquer uma de mais de 20 outras linguagens compatíveis com CTS.
- Um único assembly pode conter vários módulos escritos em diferentes linguagens .NET e os tipos podem fazer referência uns aos outros como se tivessem sido escritos na mesma linguagem.

Arquitetura .NET

- Além dos serviços de tempo de execução, o .NET também inclui bibliotecas extensas.
- Essas bibliotecas suportam muitas cargas de trabalho diferentes.
- Eles são organizados em namespaces que fornecem uma ampla variedade de funcionalidades úteis para tudo, desde entrada e saída de arquivos até manipulação de strings, análise XML, frameworks de aplicações Web e controles de Formulários do Windows.
- Uma aplicação C# típica usa a biblioteca de classes .NET extensivamente para lidar com tarefas comuns de "encanamento".

Hello World

- O programa "Hello, World" é tradicionalmente usado para introduzir uma linguagem de programação. Aqui está em C#:

C#

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Hello World

- O programa "Hello, World" começa com uma diretiva **using** que faz referência ao namespace System.
- Os namespaces fornecem um meio hierárquico de organizar programas e bibliotecas C#.
- Os namespaces contêm tipos e outros namespaces - por exemplo, o namespace System contém vários tipos, como a classe Console referenciada no programa e vários outros namespaces, como IO e Collections.
- Uma diretiva using que faz referência a um determinado namespace permite o uso não qualificado dos tipos que são membros desse namespace.
- Por causa da diretiva **using**, o programa pode usar Console.WriteLine como uma abreviação para System.Console.WriteLine.

Hello World

- A classe Hello declarada pelo programa "Hello, World" tem um único membro, o método denominado Main.
- O método Main é declarado com o modificador static.
- Embora os métodos de instância possam fazer referência a uma determinada instância de objeto envolvente usando a palavra-chave this, os métodos estáticos operam sem referência a um objeto específico.
- Por convenção, um método estático denominado Main serve como ponto de entrada de um programa C#.
- A saída do programa é produzida pelo método WriteLine da classe Console no namespace System.
- Essa classe é fornecida pelas bibliotecas de classes padrão, que, por padrão, são referenciadas automaticamente pelo compilador.

Tipos e Variáveis

- Um tipo define a estrutura e o comportamento de quaisquer dados em C#.
- A declaração de um tipo pode incluir seus membros, tipo base, interfaces que ele implementa e operações permitidas para aquele tipo.
- Uma variável é um rótulo que se refere a uma instância de um tipo específico.
- Existem dois tipos de tipos em C#:
  - **value types**
  - **reference types**.

Tipos e Variáveis

- Variáveis de value types contêm diretamente seus dados.
- Variáveis de reference types armazenam referências a seus dados, sendo os últimos conhecidos como objetos.
- Com os tipos de referência, é possível que duas variáveis façam referência ao mesmo objeto e é possível que as operações em uma variável afetem o objeto referenciado pela outra variável.
- Com os value types, cada variável tem sua própria cópia dos dados e não é possível que as operações em uma afetem a outra (exceto para variáveis de parâmetro ref e out).

Tipos e Variáveis

- Um **identificador** é um nome de variável.
- Um identificador é uma sequência de caracteres Unicode sem nenhum espaço em branco.
- Um identificador pode ser uma palavra reservada C#, se for prefixado por @.
- Usar uma palavra reservada como identificador pode ser útil ao interagir com outras linguagens.
- Os value types do C# são divididos em simple types, enum types, struct types, nullable value types, e tuple value types.
- Os reference types do C# são divididos em class types, interface types, array types, e delegate types.

- Value types
  - Simple types
    - Signed integral: sbyte, short, int, long
    - Unsigned integral: byte, ushort, uint, ulong
    - Unicode characters: char, que representa uma unidade de código UTF-16
    - IEEE binary floating-point: float, double
    - High-precision decimal floating-point: decimal
    - Boolean: bool, que representa valores booleanos - valores que são true ou false
  - Enum types
    - Tipos definidos pelo usuário na forma enum E {...}. Um tipo enum é um tipo distinto com constantes nomeadas. Cada tipo de enum tem um tipo subjacente, que deve ser um dos oito tipos integrais. O conjunto de valores de um tipo enum é igual ao conjunto de valores do tipo subjacente.
  - Struct types
    - Tipos definidos pelo usuário na forma struct S {...}
  - Nullable value types
    - Extensões de todos os outros value types com um valor null
  - Tuple value types
    - Tipos definidos pelo usuário na forma (T1, T2, ...)

- Reference types
  - Class types
    - Classe base final de todos os outros tipos: `object`
    - Strings Unicode: `string`, que representa uma sequência de unidades de código UTF-16
    - Tipos definidos pelo usuário na forma `class C {...}`
  - Interface types
    - Tipos definidos pelo usuário na forma `interface I {...}`
  - Array types
    - Unidimensional, multidimensional e jagged. Por exemplo: `int[]`, `int[,]` e `int[][]`
  - Delegate types
    - Tipos definidos pelo usuário na forma `elegate int D(...)`

Tipos e Variáveis

- Os programas C# usam declarações de tipo para criar novos tipos.
- Uma declaração de tipo especifica o nome e os membros do novo tipo.
- Seis das categorias de tipos do C# são definidas pelo usuário:
  - class types, struct types, interface types, enum types, delegate types, e tuple value types.

Tipos e Variáveis

- Um class type define uma estrutura de dados que contém membros dados (campos) e membros função (métodos, propriedades e outros). Class types suportam herança única e polimorfismo, mecanismos pelos quais as classes derivadas podem estender e especializar as classes básicas.
- Um struct type é semelhante a um class type, pois representa uma estrutura com membros dados e membros função. No entanto, ao contrário das classes, structs são value types e normalmente não requerem alocação de heap. Struct types não oferecem suporte à herança especificada pelo usuário e todos os struct types herdam implicitamente do type object.

Tipos e Variáveis

- Um interface type define um contrato como um conjunto nomeado de membros públicos. Uma class ou struct que implementa uma interface deve fornecer implementações dos membros da interface. Uma interface pode herdar de várias interfaces de base e uma class ou struct pode implementar várias interfaces.
- Um delegate type representa referências a métodos com uma lista de parâmetros específica e return type. Delegates possibilitam tratar métodos como entidades que podem ser atribuídas a variáveis e passadas como parâmetros. Delegates são análogos aos tipos de função fornecidos por linguagens funcionais. Eles também são semelhantes ao conceito de ponteiros de função encontrados em algumas outras linguagens. Ao contrário dos ponteiros de função, os delegates são orientados a objetos e com segurança de tipo.

Tipos e Variáveis

- Os class, struct, interface, and delegate types oferecem suporte a genéricos, por meio dos quais podem ser parametrizados com outros tipos.
- C# oferece suporte a arrays unidimensionais e multidimensionais de qualquer tipo. Ao contrário dos tipos listados acima, os tipos de array não precisam ser declarados antes de serem usados. Em vez disso, os tipos de array são construídos seguindo um nome de tipo com colchetes. Por exemplo, int[] é um array unidimensional de int, int[,] é um array bidimensional de int e int[][] é um array unidimensional de arrays unidimensionais ou um "jagged" array, de int.

Tipos e Variáveis

- Nullable types não requerem uma definição separada.
- Para cada tipo non-nullable T, há um tipo nullable correspondente T?, que pode conter um valor adicional, null.
- Por exemplo, int? é um tipo que pode conter qualquer número inteiro de 32 bits ou o valor null e string? é um tipo que pode conter qualquer string ou o valor null.
- O sistema de tipos do C# é unificado de forma que um valor de qualquer tipo possa ser tratado como um object. Cada tipo em C# deriva direta ou indiretamente do object class type, e object é a classe base final de todos os tipos.
- Valores de reference types são tratados como objects simplesmente visualizando os valores como type object.
- Os valores value types são tratados como objects por meio da execução de operações de **boxing** e **unboxing**.

Tipos e Variáveis

- No exemplo a seguir, um valor int é convertido em objeto e novamente em int.

C#

```
int i = 123;
object o = i;    // Boxing
int j = (int)o;  // Unboxing
```

Tipos e Variáveis

- Quando um valor de um value type é atribuído a uma referência de object, uma "box" é alocada para conter o valor.
- Essa box é uma instância de um reference type e o valor é copiado para essa box.
- Por outro lado, quando uma referência de object é convertida em um value type, é feita uma verificação de que o object referenciado é uma box do value type correto.
- Se a verificação for bem-sucedida, o valor na box será copiado para o value type.
- O sistema de tipo unificado do C# significa efetivamente que os value types são tratados como referências de object "sob demanda".
- Por causa da unificação, as bibliotecas de propósito geral que usam type object podem ser usadas com todos os tipos que derivam de object, incluindo reference types e value types.

Tipos e Variáveis

- Existem vários tipos de variáveis em C#, incluindo campos, elementos de array, variáveis locais e parâmetros.
- As variáveis representam os locais de armazenamento.
- Cada variável possui um tipo que determina quais valores podem ser armazenados na variável, conforme mostrado a seguir.
  - Non-nullable value type
  - Nullable value type
  - object
  - Class type
  - Interface type
  - Array type
  - Delegate type

Estrutura do Programa

- Os principais conceitos organizacionais em C# são:
  - **programs**, **namespaces**, **types**, **members** e **assemblies**.
- Os programas declaram types, que contêm members e podem ser organizados em namespaces.
- Classes, structs e interfaces são exemplos de types.
- Campos, métodos, propriedades e eventos são exemplos de members.
- Quando os programas C# são compilados, eles são fisicamente empacotados em assemblies. Os assemblies normalmente têm a extensão de arquivo .exe ou .dll, dependendo se implementam aplicativos ou bibliotecas, respectivamente.

- Como um pequeno exemplo, considere um assembly que contém o seguinte código:

C#

```
using System;

namespace Acme.Collections
{
    public class Stack<T>
    {
        Entry _top;

        public void Push(T data)
        {
            _top = new Entry(_top, data);
        }

        public T Pop()
        {
            if (_top == null)
            {
                throw new InvalidOperationException();
            }
            T result = _top.Data;
            _top = _top.Next;

            return result;
        }

        class Entry
        {
            public Entry Next { get; set; }
            public T Data { get; set; }

            public Entry(Entry next, T data)
            {
                Next = next;
                Data = data;
            }
        }
    }
}
```

Estrutura do Programa

- O nome totalmente qualificado desta classe é Acme.Collections.Stack.
- A classe contém vários members: um campo denominado top, dois métodos denominados Push e Pop e uma classe aninhada denominada Entry.
- A classe Entry contém ainda três members: um campo denominado next, um campo denominado data e um construtor.
- O Stack é uma classe genérica. Ele tem um parâmetro de tipo, T, que é substituído por um tipo concreto quando é usado.
- Uma stack é uma coleção "first in - last out" (FILO).
- Novos elementos são adicionados ao topo da stack. Quando um elemento é removido, ele é removido do topo da stack.
- O exemplo anterior declara o tipo Stack que define o armazenamento e o comportamento de uma stack. Você pode declarar uma variável que se refere a uma instância do tipo Stack para usar essa funcionalidade.

Estrutura do Programa

- Os assemblies contêm código executável na forma de instruções de Intermediate Language (IL) e informações simbólicas na forma de metadados.
- Antes de ser executado, o compilador Just-In-Time (JIT) do .NET Common Language Runtime converte o código IL em um assembly em código específico do processador.
- Como um assembly é uma unidade autodescritiva de funcionalidade que contém código e metadados, não há necessidade de diretivas #include e arquivos de cabeçalho em C#.
- Os types e members públicos contidos em um determinado assembly são disponibilizados em um programa C# simplesmente fazendo referência a esse assembly ao compilar o programa.

Estrutura do Programa

- Por exemplo, este programa usa a classe `Acme.Collections.Stack` do assembly `acme.dll`:

C#

```
using System;
using Acme.Collections;

class Example
{
    public static void Main()
    {
        var s = new Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

Estrutura do Programa

- Para compilar este programa, você precisaria fazer referência ao assembly que contém a classe stack definida no exemplo anterior.
- Os programas C# podem ser armazenados em vários arquivos de origem.
- Quando um programa C# é compilado, todos os arquivos de origem são processados juntos e os arquivos de origem podem fazer referência uns aos outros livremente.
- Conceitualmente, é como se todos os arquivos de origem fossem concatenados em um grande arquivo antes de serem processados.
- Declarações de encaminhamento nunca são necessárias em C# porque, com poucas exceções, a ordem das declarações é insignificante. C# não limita um arquivo de origem a declarar apenas um tipo público nem exige que o nome do arquivo de origem corresponda a um tipo declarado no arquivo de origem.

Referência

https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/