## **InterfacesAntarmuka**

An **interface** in C# is a contract that specifies a set of members (methods, properties, events, indexers) that a class or struct must implement. Unlike classes, interfaces primarily define *what* a type can do, rather than *how* it does it or *what data* it holds.Antarmuka dalam C# adalah kontrak yang menentukan kumpulan anggota (metode, properti, peristiwa, indeks) yang harus diimplementasikan oleh kelas atau struktur. Berbeda dengan kelas, antarmuka terutama mendefinisikan apa yang dapat dilakukan oleh suatu tipe, bukan bagaimana cara melakukannya atau data apa yang dimilikinya.

Key characteristics of interfaces:

* **Behavior Definition Only:** Interfaces can define only functions (methods, properties, events, indexers). They **cannot contain fields** (data) because their purpose is to specify behavior, not state. Definisi Perilaku Saja: Antarmuka hanya dapat mendefinisikan fungsi (metode, properti, peristiwa, indeks). Mereka tidak dapat mengandung bidang (data) karena tujuannya adalah untuk menentukan perilaku, bukan keadaan.
* **Implicitly Abstract Members:** Interface members are implicitly abstract by default, meaning they do not provide an implementation within the interface itself. The implementing class or struct is responsible for providing the actual code for these members.Anggota Abstrak Implisit: Anggota antarmuka secara default bersifat abstrak secara implisit, artinya mereka tidak menyediakan implementasi di dalam antarmuka itu sendiri. Kelas atau struktur yang mengimplementasikan antarmuka tersebut bertanggung jawab untuk menyediakan kode aktual untuk anggota-anggota tersebut.
* **Multiple Implementation:** A class or struct can **implement multiple interfaces**. This is a key advantage over classes, as a class can only inherit from a single base class, and structs cannot inherit at all (beyond System.ValueType). This allows a single type to embody diverse functionalities from different contracts.Implementasi Ganda: Sebuah kelas atau struct dapat mengimplementasikan beberapa antarmuka. Ini merupakan keunggulan utama dibandingkan dengan kelas, karena sebuah kelas hanya dapat mewarisi dari satu kelas dasar, dan struct tidak dapat mewarisi sama sekali (di luar System.ValueType). Hal ini memungkinkan sebuah tipe tunggal untuk mewakili berbagai fungsionalitas dari kontrak yang berbeda.
* **Implicitly public:** Interface members are always implicitly public and cannot declare an access modifier. This is because interfaces define a public contract.Secara implisit publik: Anggota antarmuka selalu secara implisit publik dan tidak dapat mendeklarasikan modifikator akses. Hal ini karena antarmuka mendefinisikan kontrak publik.

Here's an example, using the IEnumerator interface from System.Collections: Berikut ini adalah contoh penggunaan antarmuka IEnumerator dari System.Collections:

|  |
| --- |
| public interface IEnumerator // Interface Declaration {  bool MoveNext(); // Method  object Current { get; } // Read-only Property  void Reset(); // Method } |

To implement an interface, a class or struct must provide a public implementation for all of its members: Untuk mengimplementasikan antarmuka, sebuah kelas atau struktur harus menyediakan implementasi publik untuk semua anggotanya:

|  |
| --- |
| internal class Countdown : IEnumerator // Implements the IEnumerator interface {  int count = 11;   public bool MoveNext() => count-- > 0; // Implementation of MoveNext  public object Current => count; // Implementation of Current  public void Reset() { throw new NotSupportedException(); } // Implementation of Reset } |

You can implicitly cast an object to any interface it implements. This is a core aspect of polymorphism with interfaces.You can implicitly cast an object to any interface it implements. This is a core aspect of polymorphism with interfaces.

|  |
| --- |
| IEnumerator e = new Countdown(); // Casts Countdown instance to IEnumerator interface while (e.MoveNext())  Console.Write(e.Current); // Output: 109876543210 |

Even though Countdown is an internal class, its members that implement IEnumerator can be called publicly by casting an instance of Countdown to IEnumerator, provided the interface itself is accessible.Meskipun Countdown adalah kelas internal, anggota-anggotanya yang mengimplementasikan antarmuka IEnumerator dapat dipanggil secara publik dengan cara mengonversi instance Countdown menjadi IEnumerator, asalkan antarmuka itu sendiri dapat diakses.

## **Extending an Interface**

Interfaces can derive from other interfaces, allowing for the creation of interface hierarchies. The derived interface "inherits" all the members of its base interfaces.Antarmuka dapat diturunkan dari antarmuka lain, memungkinkan pembentukan hierarki antarmuka. Antarmuka yang diturunkan “mewarisi” semua anggota dari antarmuka dasarnya.

|  |
| --- |
| public interface IUndoable { void Undo(); } public interface IRedoable : IUndoable { void Redo(); } // IRedoable extends IUndoable |

Any type that implements IRedoable must also implement all members of IUndoable (in this case, Undo()).Setiap tipe yang mengimplementasikan IRedoable juga harus mengimplementasikan semua anggota dari IUndoable (dalam hal ini, Undo()).

## **Explicit Interface Implementation**

When a class implements multiple interfaces, sometimes there can be a **collision** between member signatures (e.g., two interfaces define a method with the same name but potentially different return types, or simply the same name and signature). To resolve such collisions, or to hide specialized interface members from the general public interface of the class, you can use **explicit interface implementation**.Ketika sebuah kelas mengimplementasikan beberapa antarmuka, terkadang dapat terjadi konflik antara tanda tangan anggota (misalnya, dua antarmuka mendefinisikan metode dengan nama yang sama tetapi jenis kembalian yang berbeda, atau hanya nama dan tanda tangan yang sama). Untuk menyelesaikan konflik semacam itu, atau untuk menyembunyikan anggota antarmuka khusus dari antarmuka umum kelas, Anda dapat menggunakan implementasi antarmuka secara eksplisit.

|  |
| --- |
| interface I1 { void Foo(); } interface I2 { int Foo(); }  public class Widget : I1, I2 // Implements both interfaces {  // Implicit implementation for I1.Foo()  public void Foo()  {  Console.WriteLine("Widget's implementation of I1.Foo");  }   // Explicit implementation for I2.Foo()  // Note the interface name preceding the member name, and no access modifier.  int I2.Foo()  {  Console.WriteLine("Widget's implementation of I2.Foo");  return 42;  } } |

When a member is explicitly implemented:Ketika suatu anggota secara eksplisit diimplementasikan:

* It is not public by default; it is only accessible when the object is cast to the specific interface type.Secara default, hal ini tidak bersifat publik; hal ini hanya dapat diakses ketika objek tersebut dikonversi ke tipe antarmuka spesifik.
* You cannot add an access modifier to an explicitly implemented member.Anda tidak dapat menambahkan modifikator akses ke anggota yang diimplementasikan secara eksplisit.

|  |
| --- |
| Widget w = new Widget(); w.Foo(); // Calls Widget's implicitly implemented Foo() (which is I1.Foo here) ((I1)w).Foo(); // Calls Widget's implementation of I1.Foo() ((I2)w).Foo(); // Calls Widget's explicit implementation of I2.Foo() |

Explicit implementation is also useful for hiding "boilerplate" or highly specialized interface members that are not part of the type's primary public API.Implementasi eksplisit juga berguna untuk menyembunyikan anggota antarmuka yang bersifat “boilerplate” atau sangat spesifik yang tidak termasuk dalam antarmuka publik utama tipe tersebut.

## **Implementing Interface Members Virtually**

An implicitly implemented interface member (where you just write public void Foo() { ... } in the class) is, by default, sealed. To allow subclasses to override this implementation, you must mark it as virtual or abstract in the base class.Anggota antarmuka yang diimplementasikan secara implisit (di mana Anda hanya menulis public void Foo() { ... } dalam kelas) secara default bersifat sealed. Untuk memungkinkan subkelas meng-override implementasi ini, Anda harus menandainya sebagai virtual atau abstract dalam kelas dasar.

|  |
| --- |
| public interface IUndoable { void Undo(); }  public class TextBox : IUndoable {  public virtual void Undo() => Console.WriteLine("TextBox.Undo"); // Marked virtual }  public class RichTextBox : TextBox {  public override void Undo() => Console.WriteLine("RichTextBox.Undo"); // Overrides } |

Calling the interface member through either the base class or the interface reference will correctly call the subclass's implementation due to polymorphism.

An explicitly implemented interface member (void IUndoable.Undo()) cannot be marked virtual and cannot be overridden in the usual manner. However, it can be **reimplemented** by a subclass. Memanggil anggota antarmuka melalui kelas dasar atau referensi antarmuka akan memanggil implementasi subkelas secara benar berkat polimorfisme.

Anggota antarmuka yang diimplementasikan secara eksplisit (void IUndoable.Undo()) tidak dapat ditandai sebagai virtual dan tidak dapat di-override dengan cara biasa. Namun, anggota tersebut dapat diimplementasikan ulang oleh subkelas.

## **Reimplementing an Interface in a Subclass**

A subclass can **reimplement** an interface member that has already been implemented by a base class. This "hijacks" the member's implementation when called through the interface.Sebuah subkelas dapat mengimplementasikan ulang anggota antarmuka yang telah diimplementasikan oleh kelas dasar. Hal ini “mengambil alih” implementasi anggota tersebut saat dipanggil melalui antarmuka.

|  |
| --- |
| public interface IUndoable { void Undo(); }  public class TextBox : IUndoable {  void IUndoable.Undo() => Console.WriteLine("TextBox.Undo"); // Explicit implementation }  public class RichTextBox : TextBox, IUndoable // Re-implements IUndoable {  public void Undo() => Console.WriteLine("RichTextBox.Undo"); // New implicit implementation } |

When called:

|  |
| --- |
| RichTextBox r = new RichTextBox(); r.Undo(); // Output: RichTextBox.Undo (calls the new implicit implementation) ((IUndoable)r).Undo(); // Output: RichTextBox.Undo (calls the new implementation through the interface) ((TextBox)r).Undo(); // This would lead to a compile-time error if TextBox's Undo was explicit.  // If TextBox.Undo was implicit and not virtual, this would call TextBox.Undo. |

Reimplementation is most effective and less confusing when dealing with explicitly implemented interface members in the base class. If the base class implements the member implicitly and not virtually, then calling through the base class type can lead to inconsistent behavior, which is generally undesirable.Reimplementasi paling efektif dan kurang membingungkan saat menangani anggota antarmuka yang diimplementasikan secara eksplisit di kelas dasar. Jika kelas dasar mengimplementasikan anggota tersebut secara implisit dan bukan secara virtual, maka memanggil melalui tipe kelas dasar dapat menyebabkan perilaku yang tidak konsisten, yang umumnya tidak diinginkan.

Alternatives to Reimplementation:

It's generally better to design base classes to avoid the need for reimplementation. Strategies include:Secara umum, lebih baik mendesain kelas dasar untuk menghindari kebutuhan untuk mengimplementasikan ulang. Strategi yang dapat digunakan meliputi:

* Marking implicitly implemented interface members as virtual when appropriate.Menandai anggota antarmuka yang diimplementasikan secara implisit sebagai virtual jika sesuai.
* For explicitly implemented members, create a protected virtual helper method that the explicit implementation calls. This allows subclasses to override the logic via the protected virtual method.Untuk anggota yang diimplementasikan secara eksplisit, buatlah metode bantu virtual terlindungi yang dipanggil oleh implementasi eksplisit tersebut. Hal ini memungkinkan subkelas untuk meng-override logika melalui metode virtual terlindungi tersebut.

|  |
| --- |
| public class TextBox : IUndoable {  void IUndoable.Undo() => Undo(); // Calls the protected virtual method  protected virtual void Undo() => Console.WriteLine("TextBox.Undo"); } public class RichTextBox : TextBox {  protected override void Undo() => Console.WriteLine("RichTextBox.Undo"); // Overrides logic } |

* This pattern ensures that all paths (direct call, interface call) lead to the most derived implementation.Pola ini memastikan bahwa semua jalur (panggilan langsung, panggilan antarmuka) mengarah ke implementasi yang paling turunannya.

## **Interfaces and Boxing**

When you convert a **struct to an interface**, **boxing** occurs. A copy of the struct is placed on the heap, and the interface reference points to this boxed copy.Ketika Anda mengonversi sebuah struct menjadi interface, proses boxing terjadi. Salinan dari struct tersebut ditempatkan di heap, dan referensi interface mengacu pada salinan yang telah di-boxing tersebut.

|  |
| --- |
| interface I { void Foo(); } struct S : I { public void Foo() { Console.WriteLine("S.Foo"); } }  S s = new S(); s.Foo(); // No boxing: Foo is called directly on the stack-allocated struct.  I i = s; // Boxing occurs here: 's' is copied to the heap, and 'i' refers to the boxed copy. i.Foo(); // Called on the boxed copy. |

Calling an implicitly implemented member directly on a struct instance does not cause boxing. However, accessing it via an interface variable *does* cause boxing.Memanggil anggota yang diimplementasikan secara implisit secara langsung pada instance struct tidak menyebabkan boxing. Namun, mengaksesnya melalui variabel antarmuka menyebabkan boxing.

## **Default Interface Members (C# 8+)**

From C# 8, you can provide a **default implementation** for an interface member. This makes the member optional for implementing classes or structs to implement.Mulai dari C# 8, Anda dapat menyediakan implementasi default untuk anggota antarmuka. Hal ini membuat anggota tersebut opsional bagi kelas atau struktur yang mengimplementasikannya.

|  |
| --- |
| interface ILogger {  void Log(string text) => Console.WriteLine(text); // Default implementation } |

This feature is highly advantageous for evolving interfaces in existing libraries without introducing breaking changes. If a class implements ILogger but does not provide its own Log method, it can still use the default.Fitur ini sangat menguntungkan untuk mengembangkan antarmuka dalam perpustakaan yang sudah ada tanpa menimbulkan perubahan yang merusak. Jika sebuah kelas mengimplementasikan ILogger tetapi tidak menyediakan metode Log-nya sendiri, kelas tersebut masih dapat menggunakan metode default.

Default implementations are always **explicit**. If a class implementing ILogger fails to define a Log method, the only way to call the default implementation is by casting the instance to the interface:Implementasi default selalu eksplisit. Jika sebuah kelas yang mengimplementasikan ILogger tidak mendefinisikan metode Log, satu-satunya cara untuk memanggil implementasi default adalah dengan melakukan casting instance ke antarmuka:

|  |
| --- |
| class Logger : ILogger { } // Does not implement Log()  ((ILogger)new Logger()).Log("message"); // Calls the default interface implementation |

This design prevents ambiguities when a class implements multiple interfaces that might provide the same default member.Desain ini mencegah ambiguitas ketika sebuah kelas mengimplementasikan beberapa antarmuka yang mungkin menyediakan anggota default yang sama.

## **Static Interface Members (C# 8+, C# 11+)**

Interfaces can also declare **static members**. There are two main kinds:Antarmuka juga dapat mendeklarasikan anggota statis. Ada dua jenis utama:

### **Static Non-Virtual Interface Members**

These exist primarily to support default interface members and are not implemented by classes or structs. Instead, they are consumed directly through the interface type. They can include fields, methods, properties, events, and indexers.Elemen-elemen ini terutama berfungsi untuk mendukung anggota antarmuka default dan tidak diimplementasikan oleh kelas atau struktur. Sebaliknya, mereka digunakan langsung melalui tipe antarmuka. Elemen-elemen ini dapat mencakup bidang, metode, properti, peristiwa, dan indeks.

|  |
| --- |
| interface ILogger {  void Log(string text) => Console.WriteLine(Prefix + text); // Uses static field  static string Prefix = ""; // Static field in interface }  ILogger.Prefix = "File log: "; // Accessing static member directly on the interface |

Static non-virtual interface members are public by default, but you can restrict their accessibility with modifiers (private, protected, internal). Instance fields are still prohibited in interfaces, maintaining the principle of defining behavior, not state.Anggota antarmuka statis non-virtual secara default bersifat publik, tetapi Anda dapat membatasi aksesibilitasnya dengan modifikator (private, protected, internal). Bidang instance masih dilarang dalam antarmuka, menjaga prinsip mendefinisikan perilaku, bukan keadaan.

### **Static Virtual/Abstract Interface Members (C# 11+)**

From C# 11, **static virtual/abstract interface members** enable **static polymorphism**, an advanced feature.Mulai dari C# 11, anggota antarmuka statis virtual/abstrak memungkinkan polimorfisme statis, fitur lanjutan.

* **static abstract**: Must be implemented by the implementing class or struct.Abstrak statis: Harus diimplementasikan oleh kelas atau struktur yang mengimplementasikannya.
* **static virtual**: Can optionally be implemented by the implementing class or struct, providing a default.static virtual: Dapat diimplementasikan secara opsional oleh kelas atau struktur yang mengimplementasikannya, dengan menyediakan nilai default.

|  |
| --- |
| interface ITypeDescribable {  static abstract string Description { get; } // Must be implemented  static virtual string Category => null; // Optional implementation }  class CustomerTest : ITypeDescribable {  public static string Description => "Customer tests"; // Mandatory implementation  public static string Category => "Unit testing"; // Optional implementation } |

These are used with **constrained generic type parameters** to allow generic algorithms to operate on static members of types that implement the interface. This is a powerful feature that underpins concepts like "Generic Math."Ini digunakan dengan parameter tipe generik yang dibatasi untuk memungkinkan algoritma generik beroperasi pada anggota statis dari tipe yang mengimplementasikan antarmuka. Ini adalah fitur yang sangat berguna yang mendasari konsep seperti “Generic Math.”

**Writing a Class Versus an Interface**

Deciding whether to use a class or an interface is a fundamental design choice:Memutuskan apakah akan menggunakan kelas atau antarmuka adalah pilihan desain yang mendasar:

* **Use classes and subclasses for types that naturally share an implementation.** This aligns with the "is a" relationship (e.g., a Dog is an Animal). If types share common data and behavior, inheritance is appropriate.Gunakan kelas dan subkelas untuk tipe yang secara alami berbagi implementasi. Hal ini sesuai dengan hubungan “adalah” (misalnya, seekor Anjing adalah seekor Hewan). Jika tipe-tipe tersebut berbagi data dan perilaku yang sama, pewarisan adalah pilihan yang tepat.
* **Use interfaces for types that have independent implementations but share a common *contract* or *capability*.** This aligns with the "can do" or "has a" relationship (e.g., a Car can be IDriveable, a Printer can be IPrintable). Different types might implement the same interface in vastly different ways.Gunakan antarmuka untuk jenis-jenis yang memiliki implementasi independen tetapi berbagi kontrak atau kemampuan yang sama. Hal ini sesuai dengan hubungan “dapat melakukan” atau “memiliki” (misalnya, sebuah Mobil dapat menjadi IDriveable, sebuah Printer dapat menjadi IPrintable). Jenis-jenis yang berbeda mungkin mengimplementasikan antarmuka yang sama dengan cara yang sangat berbeda.

Consider the example of animals:Pertimbangkan contoh hewan:

If Bird, Insect, FlyingCreature, and Carnivore were all classes, you couldn't represent an Eagle as a Bird, FlyingCreature, and Carnivore using class inheritance due to C#'s single-inheritance rule.Jika Bird, Insect, FlyingCreature, dan Carnivore semuanya adalah kelas, Anda tidak dapat mewakili seekor Elang sebagai Bird, FlyingCreature, dan Carnivore menggunakan pewarisan kelas karena aturan pewarisan tunggal dalam C#.

The solution is to convert capabilities that can be implemented in independent ways into interfaces:Solusinya adalah mengubah kemampuan yang dapat diimplementasikan secara mandiri menjadi antarmuka:

|  |
| --- |
| abstract class Animal {} abstract class Bird : Animal {} abstract class Insect : Animal {}  interface IFlyingCreature {} // Different creatures fly differently interface ICarnivore {} // Different creatures hunt/eat differently  // Concrete classes can now implement multiple interfaces: class Ostrich : Bird {} class Eagle : Bird, IFlyingCreature, ICarnivore {} // OK class Bee : Insect, IFlyingCreature {} // OK class Flea : Insect, ICarnivore {} // OK |

This allows Eagle to be a Bird (sharing core animal and bird implementations) and also have the capabilities of IFlyingCreature and ICarnivore (implemented distinctly). Interfaces are essential for achieving flexibility, loose coupling, and plug-in architectures in your C# applications.Hal ini memungkinkan Eagle menjadi seekor burung (membagikan implementasi inti hewan dan burung) sekaligus memiliki kemampuan IFlyingCreature dan ICarnivore (diimplementasikan secara terpisah). Antarmuka sangat penting untuk mencapai fleksibilitas, ketergantungan yang longgar, dan arsitektur plug-in dalam aplikasi C# Anda.