## **Inheritance**

**Inheritance** is a mechanism where a new class (called a **derived class** or **subclass**) can acquire the properties and methods of an existing class (called a **base class** or **superclass**). This means the derived class "inherits" functionality from the base class, allowing you to extend or customize it without writing everything from scratch.

Pewarisan adalah mekanisme di mana kelas baru (disebut kelas turunan atau subkelas) dapat memperoleh properti dan metode dari kelas yang sudah ada (disebut kelas dasar atau superkelas). Ini berarti kelas turunan “mewarisi” fungsionalitas dari kelas dasar, sehingga Anda dapat memperluas atau menyesuaikannya tanpa menulis semuanya dari awal.

In C#, a class can inherit from only a **single base class**, but a base class can be inherited by many derived classes, forming a class hierarchy.

Dalam C#, sebuah kelas dapat mewarisi hanya dari satu kelas dasar, tetapi sebuah kelas dasar dapat diwarisi oleh banyak kelas turunan, membentuk sebuah hirarki kelas.

Consider this example:

|  |
| --- |
| public class Asset // Base Class {  public string Name; }  // Stock inherits from Asset public class Stock : Asset // The colon (:) signifies inheritance {  public long SharesOwned; }  // House also inherits from Asset public class House : Asset {  public decimal Mortgage; } |

Here, Stock and House are derived classes, and Asset is the base class. Both Stock and House automatically get the Name field from Asset, in addition to their own unique members.

Di sini, Stock dan House adalah kelas turunan, dan Asset adalah kelas dasar. Baik Stock dan House secara otomatis mendapatkan field Name dari Asset, selain anggota uniknya masing-masing.

|  |
| --- |
| Stock msft = new Stock { Name = "MSFT", SharesOwned = 1000 }; Console.WriteLine(msft.Name); // Output: MSFT (inherited from Asset) Console.WriteLine(msft.SharesOwned); // Output: 1000  House mansion = new House { Name = "Mansion", Mortgage = 250000 }; Console.WriteLine(mansion.Name); // Output: Mansion (inherited from Asset) Console.WriteLine(mansion.Mortgage); // Output: 250000 |

## **Polymorphism**

**Polymorphism** means "many forms." In object-oriented programming, it refers to the ability of a variable of a base class type to refer to an object of any of its derived (sub)classes.

For instance, a method that accepts an Asset parameter can work with any object that *is* an Asset, including Stock and House instances, because Stock and House are types of Asset.

Polimorfisme berarti "banyak bentuk". Dalam pemrograman berorientasi objek, hal ini merujuk pada kemampuan variabel dari tipe kelas dasar untuk merujuk pada objek dari salah satu turunan (sub) kelasnya.

Misalnya, sebuah metode yang menerima parameter Aset dapat bekerja dengan objek apa pun yang merupakan Aset, termasuk contoh Saham dan Rumah, karena Saham dan Rumah adalah jenis Aset.

|  |
| --- |
| public static void Display(Asset asset) // Accepts an Asset or any of its subclasses {  System.Console.WriteLine(asset.Name); }  Stock msft = new Stock { Name = "MSFT", SharesOwned = 1000 }; House mansion = new House { Name = "Mansion", Mortgage = 250000 };  Display(msft); // Calls Display with a Stock object Display(mansion); // Calls Display with a House object |

This works because subclasses (Stock, House) possess all the features of their base class (Asset). However, the reverse is not true: you cannot pass a general Asset object to a method specifically expecting a House, because an Asset object might not have House-specific members like Mortgage.

Hal ini bekerja karena subkelas (Stock, House) memiliki semua fitur dari kelas dasarnya (Asset). Namun, kebalikannya tidak benar: Anda tidak dapat mengoper objek Aset umum ke metode yang secara khusus mengharapkan Rumah, karena objek Aset mungkin tidak memiliki anggota khusus Rumah seperti Hipotek.

|  |
| --- |
| public static void Display(House house) // Specifically expects a House {  System.Console.WriteLine(house.Mortgage); }  // Display(new Asset()); // Compile-time error: An Asset is not guaranteed to be a House |

## **Casting and Reference Conversions**

Object references can be converted between compatible types in a hierarchy through **reference conversions**. This doesn't alter the underlying object; it just changes the "view" that the reference variable has of that object.

Referensi objek dapat dikonversi di antara tipe-tipe yang kompatibel dalam hierarki melalui konversi referensi. Hal ini tidak mengubah objek yang mendasarinya; ini hanya mengubah “pandangan” yang dimiliki variabel referensi terhadap objek tersebut.

### **Upcasting**

An **upcast** operation creates a base class reference from a subclass reference. This conversion is always **implicit** and always **succeeds** because a subclass *is a* type of its base class.

Operasi upcast membuat referensi kelas dasar dari referensi subkelas. Konversi ini selalu implisit dan selalu berhasil karena subkelas adalah tipe dari kelas dasarnya.

|  |
| --- |
| Stock msft = new Stock(); Asset a = msft; // Upcast: 'a' now refers to the same Stock object as 'msft' |

After the upcast, a and msft refer to the identical object (Console.WriteLine(a == msft); would output True). However, a (being an Asset type) has a more restricted view of the object; it can only access members defined in the Asset class.

Setelah upcast, a dan msft merujuk pada objek yang sama (Console.WriteLine(a == msft); akan menghasilkan True). Namun, a (sebagai tipe Aset) memiliki pandangan yang lebih terbatas terhadap objek; ia hanya dapat mengakses anggota yang didefinisikan dalam kelas Aset.

|  |
| --- |
| Console.WriteLine(a.Name); // OK: Name is defined in Asset // Console.WriteLine(a.SharesOwned); // Compile-time error: SharesOwned is a Stock-specific member |

### **Downcasting**

A **downcast** operation creates a subclass reference from a base class reference. This conversion requires an **explicit cast** because it can potentially fail at runtime if the object being referenced is not actually an instance of the target subclass.

Operasi penurunan menciptakan referensi subkelas dari referensi kelas dasar. Konversi ini membutuhkan cast eksplisit karena berpotensi gagal pada saat runtime jika objek yang direferensikan bukan merupakan instance dari subkelas target.

|  |
| --- |
| Stock msft = new Stock(); Asset a = msft; // Upcast Stock s = (Stock)a; // Downcast: 's' now refers to the same Stock object Console.WriteLine(s.SharesOwned); // OK Console.WriteLine(s == a); // True Console.WriteLine(s == msft); // True |

If a downcast is attempted on an object that is not of the target type (or a derived type), an InvalidCastException is thrown at runtime.

Jika downcast dicoba pada objek yang bukan tipe target (atau tipe turunan), InvalidCastException akan dilemparkan pada saat runtime.

|  |
| --- |
| House h = new House(); Asset a = h; // Upcast // Stock s = (Stock)a; // This would throw an InvalidCastException at runtime  // because 'a' currently refers to a House object, not a Stock. |

### **The as Operator**

The **as operator** performs a downcast but provides a safer alternative to a direct cast. If the downcast fails, it evaluates to null instead of throwing an InvalidCastException.

Operator as melakukan downcast tetapi menyediakan alternatif yang lebih aman daripada cast langsung. Jika downcast gagal, maka akan dievaluasi menjadi null dan bukannya melempar InvalidCastException.

|  |
| --- |
| Asset a = new Asset(); Stock s = a as Stock; // s will be null because 'a' is just an Asset, not a Stock. No exception. |

This is particularly useful when you intend to check for null afterward:

Hal ini sangat berguna ketika Anda berniat untuk memeriksa null setelahnya:

|  |
| --- |
| if (s != null)  Console.WriteLine(s.SharesOwned); |

**Caution:** If you don't check for null and the as operator returns null, subsequent operations on s (like accessing SharesOwned) will result in a NullReferenceException, which can be more ambiguous than InvalidCastException. Use direct casts when you are *certain* of the type and want an exception if you are wrong; use as when you are *uncertain* and want to handle the possibility of failure gracefully.

Jika Anda tidak memeriksa null dan operator as mengembalikan null, operasi selanjutnya pada s (seperti mengakses SharesOwned) akan menghasilkan NullReferenceException, yang bisa lebih ambigu daripada InvalidCastException. Gunakan cast langsung ketika Anda yakin dengan jenisnya dan menginginkan pengecualian jika Anda salah; gunakan as ketika Anda tidak yakin dan ingin menangani kemungkinan kegagalan dengan baik.

The as operator cannot be used for custom or numeric conversions. Operator as tidak dapat digunakan untuk konversi kustom atau numerik.

### **The is Operator**

The **is operator** tests whether a variable matches a pattern, most commonly a **type pattern**. It checks if a reference conversion would succeed, essentially asking: "Is this object an instance of (or derived from) this specific type, or does it implement this interface?"

Operator is menguji apakah sebuah variabel cocok dengan sebuah pola, yang paling umum adalah pola tipe. Operator ini memeriksa apakah konversi referensi akan berhasil, yang pada dasarnya menanyakan: “Apakah objek ini merupakan contoh dari (atau turunan dari) tipe spesifik ini, atau apakah objek ini mengimplementasikan antarmuka ini?”

|  |
| --- |
| Asset a = new Stock(); // 'a' actually holds a Stock object if (a is Stock) // Checks if 'a' can be successfully cast to Stock  Console.WriteLine(((Stock)a).SharesOwned); // This block executes |

The is operator also works for unboxing conversions but not for custom or numeric conversions.

Operator is juga berfungsi untuk konversi tanpa kotak tetapi tidak untuk konversi kustom atau numerik.

#### **Introducing a Pattern Variable (C# 7+)**

From C# 7, you can combine the is operator with a variable declaration, creating a **pattern variable**. If the is check is successful, the variable is automatically cast and assigned the value, making the code more concise.

Dari C# 7, Anda dapat menggabungkan operator is dengan deklarasi variabel, membuat variabel pola. Jika pemeriksaan is berhasil, variabel akan secara otomatis di-cast dan diberi nilai, sehingga membuat kode menjadi lebih ringkas

|  |
| --- |
| if (a is Stock s) // If 'a' is a Stock, assign it to a new 's' variable  Console.WriteLine(s.SharesOwned); // 's' is now available and correctly typed |

This is equivalent to:

|  |
| --- |
| Stock s; if (a is Stock) {  s = (Stock)a;  Console.WriteLine(s.SharesOwned); } |

The pattern variable s remains in scope outside the is expression and can be used in subsequent && conditions.

Variabel pola s tetap berada dalam cakupan di luar ekspresi is dan dapat digunakan dalam kondisi && berikutnya.

## **Virtual Function Members**

**Virtual function members** allow a base class to define a method (or property, indexer, event) that its subclasses can choose to **override** and provide their own specialized implementation.

Anggota fungsi virtual memungkinkan kelas dasar mendefinisikan metode (atau properti, pengindeks, peristiwa) yang dapat ditimpa oleh subkelasnya dan menyediakan implementasi khusus mereka sendiri.

|  |
| --- |
| public class Asset {  public string Name;  public virtual decimal Liability => 0; // Virtual property with a default implementation } |

A subclass overrides a virtual member by using the **override modifier**:

Subkelas menimpa anggota virtual dengan menggunakan pengubah timpa:

|  |
| --- |
| public class Stock : Asset {  public long SharesOwned;  // Stock doesn't need to override Liability, it uses Asset's default (0). }  public class House : Asset {  public decimal Mortgage;  public override decimal Liability => Mortgage; // House overrides Liability } |

Now, when accessing Liability(kewajiban):

|  |
| --- |
| House mansion = new House { Name = "McMansion", Mortgage = 250000 }; Asset a = mansion;  Console.WriteLine(mansion.Liability); // Output: 250000 (House's overridden implementation) Console.WriteLine(a.Liability); // Output: 250000 (Polymorphism: Calls House's implementation) |

The signatures, return types, and accessibility of the virtual and overridden methods must match. An overridden method can call its base class implementation using the base keyword.

Tanda tangan, tipe pengembalian, dan aksesibilitas metode virtual dan metode yang ditimpa harus sesuai. Metode yang ditimpa dapat memanggil implementasi kelas dasarnya dengan menggunakan kata kunci dasar.

**Caution:** Calling virtual methods from a constructor can be dangerous. Subclasses might override the method, and their overridden implementation might try to access fields that have not yet been initialized by the constructor, leading to unexpected behavior or errors.

Memanggil metode virtual dari konstruktor bisa berbahaya. Subkelas dapat menimpa metode, dan implementasi yang ditimpa dapat mencoba mengakses bidang yang belum diinisialisasi oleh konstruktor, yang mengarah ke perilaku yang tidak diharapkan atau kesalahan.

### **Covariant Return Types (C# 9+)**

From C# 9, you can override a method (or a property's get accessor) and specify a **more derived (subclassed) return type** than the base method. This is called **covariant return types**.

Dari C# 9, Anda bisa mengganti sebuah metode (atau pengakses get sebuah properti) dan menentukan tipe pengembalian yang lebih turunan (subkelas) dari metode dasar. Hal ini disebut tipe pengembalian kovarian.

|  |
| --- |
| public class Asset {  public string Name;  public virtual Asset Clone() => new Asset { Name = Name }; // Returns Asset }  public class House : Asset {  public decimal Mortgage;  public override House Clone() => new House // Returns House (which is an Asset)  { Name = Name, Mortgage = Mortgage }; } |

This is allowed because returning a House (a subclass of Asset) still fulfills the contract that Clone must return an Asset. This feature reduces the need for explicit downcasting when using overridden methods.

Hal ini diperbolehkan karena mengembalikan House (subkelas dari Aset) masih memenuhi kontrak bahwa Clone harus mengembalikan Aset. Fitur ini mengurangi kebutuhan untuk melakukan downcasting secara eksplisit ketika menggunakan metode override.

## **Abstract Classes and Abstract Members**

An **abstract class** is a base class that cannot be instantiated directly. It serves as a blueprint for other classes and can define **abstract members**.

**Abstract members** are like virtual members, but they provide *no* default implementation in the abstract class. They simply declare that a subclass *must* provide an implementation. A subclass that inherits an abstract member must either provide an override implementation or itself be declared abstract.

Kelas abstrak adalah kelas dasar yang tidak dapat diinstansiasi secara langsung. Kelas ini berfungsi sebagai cetak biru untuk kelas lain dan dapat mendefinisikan anggota abstrak.

Anggota abstrak seperti anggota virtual, tetapi mereka tidak menyediakan implementasi default dalam kelas abstrak. Mereka hanya menyatakan bahwa sebuah subkelas harus menyediakan implementasi. Subkelas yang mewarisi anggota abstrak harus menyediakan implementasi override atau dengan sendirinya dinyatakan abstrak.

|  |
| --- |
| public abstract class Asset // Abstract class - cannot be instantiated {  public string Name;  public abstract decimal NetValue { get; } // Abstract property - no implementation here }  public class Stock : Asset {  public long SharesOwned;  public decimal CurrentPrice;  public override decimal NetValue => CurrentPrice \* SharesOwned; // Must provide implementation } |

## **Hiding Inherited Members**

If a base class and a derived class define members with **identical names and signatures**, the derived class member is said to **hide** the base class member. This often happens by accident. The compiler issues a warning and resolves the ambiguity based on the compile-time type of the variable:

* References to the base class type bind to the base class member.
* References to the derived class type bind to the derived class member.

Jika kelas dasar dan kelas turunan mendefinisikan anggota dengan nama dan tanda tangan yang sama, anggota kelas turunan dikatakan menyembunyikan anggota kelas dasar. Hal ini sering terjadi secara tidak sengaja. Kompiler mengeluarkan peringatan dan menyelesaikan ambiguitas berdasarkan tipe waktu kompilasi variabel:

* Referensi ke tipe kelas dasar mengikat anggota kelas dasar.
* Referensi ke tipe kelas turunan mengikat ke anggota kelas turunan.

|  |
| --- |
| public class A { public int Counter = 1; } public class B : A { public int Counter = 2; } // B.Counter hides A.Counter  A a = new B(); Console.WriteLine(a.Counter); // Output: 1 (A.Counter is accessed because 'a' is of type A)  B b = new B(); Console.WriteLine(b.Counter); // Output: 2 (B.Counter is accessed because 'b' is of type B) |

If you intend to hide a member, you can apply the **new modifier** to the member in the subclass. This simply suppresses the compiler warning, making your intent explicit.

Jika Anda berniat menyembunyikan sebuah anggota, Anda bisa menerapkan pengubah baru pada anggota tersebut di dalam subkelas. Hal ini akan menyembunyikan peringatan dari kompiler, sehingga maksud Anda menjadi eksplisit.

|  |
| --- |
| public class A { public int Counter = 1; } public class B : A { public new int Counter = 2; } // 'new' indicates intentional hiding |

### **new versus override**

It's crucial to understand the difference between new (hiding) and override (polymorphism).

Sangat penting untuk memahami perbedaan antara new (menyembunyikan) dan menimpa (polimorfisme).

|  |
| --- |
| public class BaseClass {  public virtual void Foo() { Console.WriteLine("BaseClass.Foo"); } }  public class Overrider : BaseClass {  public override void Foo() { Console.WriteLine("Overrider.Foo"); } // Overrides virtual method }  public class Hider : BaseClass {  public new void Foo() { Console.WriteLine("Hider.Foo"); } // Hides base method } |

Behavior comparison:

|  |
| --- |
| Overrider over = new Overrider(); BaseClass b1 = over; over.Foo(); // Output: Overrider.Foo (direct call) b1.Foo(); // Output: Overrider.Foo (polymorphism applies)  Hider h = new Hider(); BaseClass b2 = h; h.Foo(); // Output: Hider.Foo (direct call) b2.Foo(); // Output: BaseClass.Foo (hiding: polymorphism does NOT apply for 'new' methods) |

When Foo is hidden with new, the method called depends on the *compile-time type* of the reference (BaseClass or Hider), not the runtime type of the object. When Foo is overriden, the method called always depends on the *runtime type* of the object.

Ketika Foo disembunyikan dengan new, metode yang dipanggil bergantung pada tipe waktu kompilasi referensi (BaseClass atau Hider), bukan tipe waktu berjalan objek. Ketika Foo ditimpa, metode yang dipanggil selalu bergantung pada tipe runtime objek.

## **Sealing Functions and Classes**

The **sealed keyword** prevents further inheritance or overriding.

* **Sealing a function member**: An overridden virtual function member can be sealed to prevent any further subclasses from overriding it.

Kata kunci yang disegel mencegah pewarisan atau penimpaan lebih lanjut.

* Menyegel anggota fungsi: Anggota fungsi virtual yang ditimpa dapat disegel untuk mencegah subkelas lebih lanjut menimpanya.

|  |
| --- |
| public class House : Asset {  public decimal Mortgage;  public sealed override decimal Liability => Mortgage; // Cannot be overridden by House's subclasses } |

* **Sealing a class**: You can apply the sealed modifier to the class itself to prevent any other classes from inheriting from it. This is more common than sealing individual members.
* Menyegel sebuah kelas: Anda bisa menerapkan pengubah tersegel ke kelas itu sendiri untuk mencegah kelas lain mewarisi darinya. Hal ini lebih umum dilakukan daripada menyegel anggota individu.

|  |
| --- |
| public sealed class FinalClass // Cannot be inherited {  // ... } |

* Note: You can seal a function member against overriding, but you cannot seal a member against being hidden (with new).
* Catatan: Anda dapat menyegel anggota fungsi agar tidak ditimpa, tetapi Anda tidak dapat menyegel anggota agar tidak disembunyikan (dengan new).

## **The base Keyword**

The **base keyword** is similar to this but refers to the immediate base class. It has two main uses:

1. **Accessing an overridden function member from the subclass**:

Kata kunci dasar mirip dengan ini, tetapi merujuk ke kelas dasar langsung. Kata kunci ini memiliki dua kegunaan utama:

1. Mengakses anggota fungsi yang ditimpa dari subkelas:

|  |
| --- |
| public class House : Asset {  public decimal Mortgage;  public override decimal Liability => base.Liability + Mortgage; // Calls Asset's Liability (which is 0) } |

1. Here, base.Liability ensures that Asset's implementation of Liability is accessed non-virtually, regardless of the object's runtime type.

2. Di sini, base.Liability memastikan bahwa implementasi Liability pada Aset diakses secara non-virtual, terlepas dari jenis runtime objek.

1. **Calling a base-class constructor**.

3. Memanggil konstruktor kelas dasar.

## **Constructors and Inheritance**

When a derived class is instantiated, its constructor runs, but it *must also* ensure that a base class constructor is called to initialize the base part of the object. Base class constructors are not automatically inherited.

Ketika kelas turunan diinstansiasi, konstruktornya akan berjalan, tetapi juga harus memastikan bahwa konstruktor kelas dasar dipanggil untuk menginisialisasi bagian dasar objek. Konstruktor kelas dasar tidak diwariskan secara otomatis.

|  |
| --- |
| public class Baseclass {  public int X;  public Baseclass() { }  public Baseclass(int x) => X = x; }  public class Subclass : Baseclass { } // No explicit constructor in Subclass  // Subclass s = new Subclass(123); // ILLEGAL: Subclass doesn't have a constructor taking an int |

To expose constructors with parameters, Subclass must redefine them and explicitly call a base class constructor using the **base keyword**:

Untuk mengekspos konstruktor dengan parameter, Subkelas harus mendefinisikan ulang dan secara eksplisit memanggil konstruktor kelas dasar menggunakan kata kunci dasar:

|  |
| --- |
| public class Subclass : Baseclass {  public Subclass(int x) : base(x) { } // Calls Baseclass(int x) constructor } |

The base keyword here works like this for chaining constructors within the same class, but it specifically calls a constructor in the base class. **Base-class constructors always execute first**, ensuring that the base part of the object is initialized before the derived part.

Kata kunci base di sini berfungsi seperti ini untuk merantai konstruktor di dalam kelas yang sama, tetapi secara khusus memanggil konstruktor di kelas dasar. Konstruktor kelas dasar selalu dieksekusi terlebih dahulu, memastikan bahwa bagian dasar dari objek diinisialisasi sebelum bagian turunannya.

### **Implicit Calling of the Parameterless Base-Class Constructor**

If a constructor in a derived class omits the base keyword, the base type's **parameterless constructor is implicitly called**.  
  
Pemanggilan Implisit dari Konstruktor Kelas Dasar Tanpa Parameter

Jika sebuah konstruktor pada kelas turunan menghilangkan kata kunci dasar, maka konstruktor tanpa parameter dari tipe dasar akan dipanggil secara implisit.

|  |
| --- |
| public class Baseclass {  public int X;  public Baseclass() { X = 1; } }  public class Subclass : Baseclass {  public Subclass() { Console.WriteLine(X); } // Implicitly calls Baseclass(), so X is 1 } |

If the base class has *no accessible parameterless constructor* (e.g., only a constructor with parameters), then all subclasses are *forced* to explicitly use the base keyword in their constructors to call one of the base class's available constructors.

Jika kelas dasar tidak memiliki konstruktor tanpa parameter yang dapat diakses (misalnya, hanya konstruktor dengan parameter), maka semua subkelas dipaksa untuk secara eksplisit menggunakan kata kunci dasar pada konstruktor mereka untuk memanggil salah satu konstruktor yang tersedia pada kelas dasar.

### **Required Members (C# 11+)**

The need for subclasses to invoke complex base class constructors can be cumbersome. C# 11 introduced **required members** to help with this. You can mark a field or property as required, meaning it must be populated via an object initializer during construction.

Kebutuhan akan subkelas untuk memanggil konstruktor kelas dasar yang kompleks dapat menjadi rumit. C# 11 memperkenalkan anggota yang diperlukan untuk membantu dalam hal ini. Anda bisa menandai sebuah field atau properti sebagai anggota yang diperlukan, yang berarti harus diisi melalui inisialisasi objek selama konstruksi.

|  |
| --- |
| public class Asset {  public required string Name; // Must be set via object initializer }  Asset a1 = new Asset { Name = "House" }; // OK // Asset a2 = new Asset(); // Error: 'Name' is required but not set |

If you *do* provide a constructor, you can use the [SetsRequiredMembers] attribute to tell the compiler that this constructor handles the required members, bypassing the object initializer restriction for that specific constructor.

|  |
| --- |
| public class Asset {  public required string Name;  public Asset() { } // This constructor requires 'Name' to be set by object initializer  [System.Diagnostics.CodeAnalysis.SetsRequiredMembers]  public Asset(string n) => Name = n; // This constructor fulfills the 'required' contract } |

This allows consumers to use either object initializers or a parameterized constructor. If the base class has a parameterless constructor, subclasses are not burdened with re-implementing all base class constructors.

Hal ini memungkinkan konsumen untuk menggunakan inisialisasi objek atau konstruktor berparameter. Jika kelas dasar memiliki konstruktor tanpa parameter, subkelas tidak dibebani untuk mengimplementasikan ulang semua konstruktor kelas dasar.

### **Constructor and Field Initialization Order**

When an object is instantiated in an inheritance hierarchy, initialization occurs in a specific order:

1. **From subclass to base class:** a. Fields in the subclass are initialized. b. Arguments to the base-class constructor call (e.g., in : base(x + 1)) are evaluated.
2. **From base class to subclass:** a. The base class's fields are initialized. b. The base class's constructor body executes. c. The derived class's constructor body executes.

Ketika sebuah objek diinstansiasi dalam hirarki pewarisan, inisialisasi terjadi dalam urutan tertentu:

1. Dari subkelas ke kelas dasar: a. Bidang dalam subkelas diinisialisasi. b. Argumen pada pemanggilan konstruktor kelas dasar (misal, in : base(x + 1)) dievaluasi.

2. Dari kelas dasar ke subkelas: a. Field-field kelas dasar diinisialisasi. b. Badan konstruktor kelas dasar dieksekusi. c. Badan konstruktor kelas turunan dieksekusi.

Example:

|  |
| --- |
| public class B {  int x = 1; // Executes 3rd  public B(int x)  {  // ... (Executes 4th)  } }  public class D : B {  int y = 1; // Executes 1st  public D(int x)  : base(x + 1) // Executes 2nd (evaluates x+1 for base constructor)  {  // ... (Executes 5th)  } } |

### **Inheritance with Primary Constructors (C# 12+)**

When using primary constructors, subclassing syntax is concise:

|  |
| --- |
| public class Baseclass (int x) { /\* ... \*/ } public class Subclass (int x, int y) : Baseclass (x) { /\* ... \*/ } // Calls base constructor |

The call Baseclass(x) is equivalent to : base(x).

## **Overloading and Resolution**

Inheritance affects how overloaded methods are resolved. When multiple overloads exist (e.g., one taking a base type and one taking a derived type), the **most specific type has precedence**.

Pewarisan mempengaruhi bagaimana metode yang kelebihan beban diselesaikan. Ketika ada beberapa kelebihan beban (misalnya, satu mengambil tipe dasar dan satu lagi mengambil tipe turunan), tipe yang paling spesifik akan didahulukan

|  |
| --- |
| static void Foo(Asset a) { Console.WriteLine("Foo(Asset)"); } static void Foo(House h) { Console.WriteLine("Foo(House)"); }  House h = new House(); Foo(h); // Calls Foo(House) - because House is more specific than Asset |

However, the specific overload to call is determined **statically at compile time**, based on the declared type of the variable, not its runtime type.

Namun, overload tertentu yang akan dipanggil ditentukan secara statis pada saat kompilasi, berdasarkan tipe yang dideklarasikan dari variabel, bukan tipe runtime.

|  |
| --- |
| Asset a = new House(); // 'a' is declared as Asset, even though it holds a House object Foo(a); // Calls Foo(Asset) - because 'a' is compile-time type Asset |

If you cast the Asset to dynamic, the overload resolution is deferred until runtime and will then be based on the object's actual type.  
Jika Anda mengubah Aset menjadi dinamis, resolusi kelebihan beban akan ditunda hingga waktu proses dan kemudian akan didasarkan pada jenis objek yang sebenarnya.

|  |
| --- |
| Asset a = new House(); Foo((dynamic)a); // Calls Foo(House) at runtime |