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.

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.

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.

```
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`.

```
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`.

```
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.

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.

```
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.

```
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.

```
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.

```
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.

```
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:

```
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.

The as operator cannot be used for custom or numeric conversions.

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?"

```
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.

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.

```
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.

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.

```
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**:

```
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`:

```
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.

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.

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**.

```
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.

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`.

```
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.

```
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.

```
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).

```
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 `override`n, the method called always depends on the *runtime type* of the object.

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.

```
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.

```
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`).

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**:

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

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

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

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.

```
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**:

```
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.

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**.

```
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.

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.

```
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.

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.

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**.

```
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.

```
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.

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