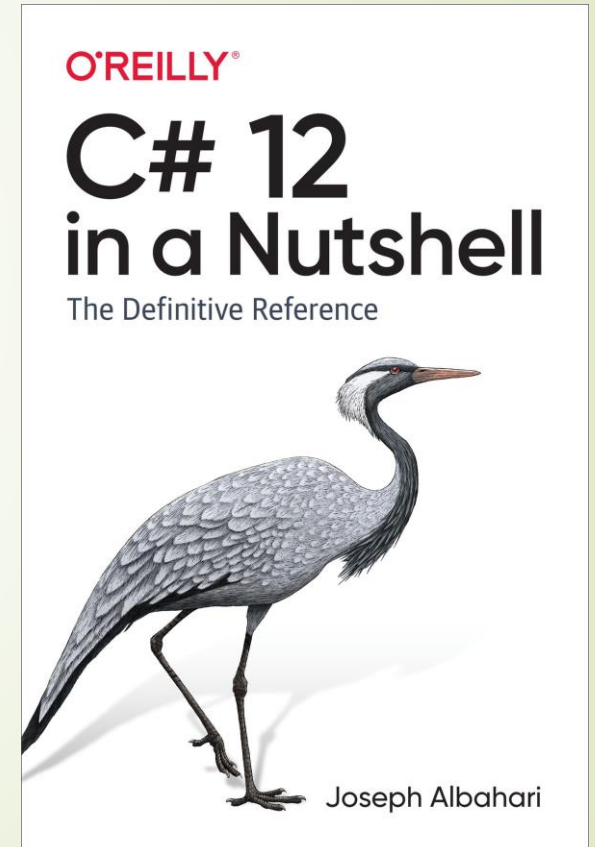


فصل سوم - مبحث وراثت

- Inheritance
- Polymorphism
- Casting and Reference Conversions
- Upcasting and Downcasting
- The as operator
- The is operator
- Virtual Function Members
- Abstract Classes and Abstract Members
- Hiding Inherited Members
- Sealing Functions and Classes
- The base Keyword
- Constructors and Inheritance



Inheritance

A class can inherit from another class to extend or customize the original class.

یه کلاس میتونه برای توسعه یا سفارشی سازی یه کلاس دیگه ازش ارث بیره.

Inheriting from a class lets you reuse the functionality in that class instead of building it from scratch.

وراثت از یه کلاس بهمون اجازه میده تا به جای اختراع مجدد چرخ، ازش استفاده کنیم و توسعه و بهبودش بدیم.

A class can inherit from only a single class but can itself be inherited by many classes, thus forming a class hierarchy.

یه کلاس میتونه فقط میتونه از یه کلاس ارث بیره ولی میتونه خودش توسط چند تا کلاس ارث برده بشه.

Inheritance

به این مثال دقت کنید:

یه کلاس داریم به اسم **Asset** که یه فیلد نام داره.

حالا دو تا کلاس **stock** و **House** رو تعریف میکنیم که از این کلاس ارث ببرند.

این دو کلاس هر آنچه کلاس **Asset** داره رو دارند و علاوه بر اون هرکدوم فیلدهای اختصاصی خودشون رو هم دارند.

```
2 references | 0 changes | 0 authors, 0 changes
public class Asset
{
    public string Name;
}

0 references | 0 changes | 0 authors, 0 changes
public class Stock : Asset // inherits from Asset
{
    public long SharesOwned;
}

0 references | 0 changes | 0 authors, 0 changes
public class House : Asset // inherits from Asset
{
    public decimal Mortgage;
}
```

Inheritance

```
Stock msft = new Stock
{
    Name = "MSFT",
    SharesOwned = 1000
};

Console.WriteLine(msft.Name);           // MSFT
Console.WriteLine(msft.SharesOwned);    // 1000

House mansion = new House
{
    Name = "Mansion",
    Mortgage = 250000
};

Console.WriteLine(mansion.Name);         // Mansion
Console.WriteLine(mansion.Mortgage);     // 250000
```

اینم نحوه استفاده از کلاس هایی که در اسلاید قبلی تعریف کردیم.

Stock و House کلاسهای مشتق شده از کلاس اصلی Asset هستند.

- به کلاس مشتق شده (derived class)، subclass هم میگویند.
- به کلاس اصلی (base class)، superclass هم میگویند.

Polymorphism

References are polymorphic. This means a variable of type x can refer to an object that subclasses x.

در کل یعنی آبجکتهایی که از جنس یه کلاس میتونه به شکل آبجکتهایی از جنس subclass های خودش در بیاد.
این مثال رو ببینید:

```
2 references | - changes | -authors, -changes
static void Display(Asset asset)
{
    System.Console.WriteLine(asset.Name);
}

Stock stock = new Stock();
House house = new House();

Display(stock);
Display(house);
```


متد display ورودی از جنس Asset قبول میکنه ولی از اونجایی که Stock و House کلاس های مشتق شده از Asset هستند میتونند به عنوان ورودی به متد display ارسال شوند.

Polymorphism

حالا اگه بیایم یه متد تعریف کنیم که ورودیش از جنس **House** باشه دیگه نمیتونیم آبجکتی از جنس **Asset** رو بهش بدیم. این یعنی کلاس های فرزند میتونند به شکل کلاس پدر در بیاند ولی برعکسش امکان پذیر نیست.

```
1 reference | - changes | -authors, -changes
static void DisplayHouse(House house)           // Will not accept Asset
{
    System.Console.WriteLine(house.Mortgage);
}

DisplayHouse(new Asset()); // Compile-time error
#endregion
```

 Asset.Asset()

CS1503: Argument 1: cannot convert from 'CSharp12Nutshell.Chapter03.Inheritance.Asset' to 'CSharp12Nutshell.Chapter03.Inheritance.House'

Show potential fixes (Ctrl+.)

Casting and Reference Conversions

An object reference can be:

- 1) Implicitly upcast to a base class reference
- 2) Explicitly downcast to a subclass reference

یک آبجکت میتونه به صورت Implicitly upcast به کلاس والد خود تبدیل بشه و

به صورت Explicitly downcast میتونه به کلاس های مشتق شده از خودش تبدیل بشه.

Upcasting and downcasting between compatible reference types performs reference conversions: a new reference is (logically) created that points to the same object. An upcast always succeeds; a downcast succeeds only if the object is suitably typed.

Upcasting

An upcast operation creates a base class reference from a subclass reference.

عملیات upcast به رفرنس جدید از کلاس اصلی ایجاد می‌کند.

طبیعی که کلاس جدید از جنس کلاس والد هست و پراپرتی های کلاس مشتق شده را ندارد.

```
Stock s = new Stock();  
Asset a = s;           // Upcast  
  
Console.WriteLine(a == s);    // True  
  
Console.WriteLine(a.Name);    // OK  
Console.WriteLine(a.SharesOwned); // Compile-time error
```

#endregion

CS1061: 'Asset' does not contain a definition for 'SharesOwned' and no accessible extension method 'SharesOwned' accepting a first argument of type 'Asset' could be found (are you missing a using directive or an assembly reference?)

Downcasting

A downcast operation creates a subclass reference from a base class reference:

عملیات **downcast** به رفرنس جدید از کلاس مشتق شده ایجاد می‌کند. این مدل تبدیل تضمینی نمیده که همیشه درست بتونه تبدیل رو انجام بده. برخی از تبدیل‌ها ممکنه هنگام کامپایل اوکی باشند ولی در **runtime** خطا بدند. خطایی هم که میدند **InvalidCastException** هست.

```
msft = new Stock();
a = msft; // Upcast
s = (Stock)a; // Downcast
Console.WriteLine(s.SharesOwned); // <No error>
Console.WriteLine(s == a); // True
Console.WriteLine(s == msft); // True

House h = new House();
Asset a2 = h; // Upcast always succeeds
Stock s2 = (Stock)a2; // Downcast fails: a is not a Stock ❌

#endregion
```

Exception Unhandled

System.InvalidCastException: 'Unable to cast object of type 'CSharp12Nutshell.Chapter03.Inheritance.House' to type 'CSharp12Nutshell.Chapter03.Inheritance.Stock'.'

[Show Call Stack](#) | [View Details](#) | [Copy Details](#) | [Start Live Share session](#)

▸ [Exception Settings](#)

The as operator


عملگر **as** در اصل یک **downcast** انجام میدهد با این تفاوت که اگر موفق نشه دیگه خطا نمیده و مقدار رو فقط **null** میکنه.

```
a = new Asset();  
s = a as Stock;           // s is null; no exception thrown
```

این عملگر فقط برای متغیرهای **reference-type** یا **nullable-type** کار میکنه:

```
long x = 3 as long;        // Compile-time error
```

```
#endregion
```

 **readonly struct System.Int64**
Represents a 64-bit signed integer.

CS0077: The as operator must be used with a reference type or nullable type ('long' is a non-nullable value type)

The is operator

عملگر **is** چک میکند که آیا متغیر با الگویی که بهش میگیریم مطابقت دارد یا نه. در سی شارپ الگوهای متنوعی میتوانیم استفاده کنیم که یکی از مهمترین آنها **type pattern** هست. مثالهای زیر رو ببینید:

همونطور که میبینید میتوانیم طوری از این عملگر استفاده کنیم که **cast** به صورت خودکار انجام شود. متغیر جدیدی که به دست می آید در خطهای بعدی میتونه استفاده بشه.

```
if (a is Stock)
    Console.WriteLine(((Stock)a).SharesOwned);

if (a is Stock ss)
    Console.WriteLine(ss.SharesOwned);

if (a is Stock s3 && s3.SharesOwned > 100000)
    Console.WriteLine("Wealthy");

if (a is Stock s4 && s4.SharesOwned > 100000)
    Console.WriteLine("Wealthy");
else
    s4 = new Stock(); // s is in scope

Console.WriteLine(s4.SharesOwned); // Still in scope
```

Virtual Function Members

یادتونه توی فصل اول در مورد `function` ها صحبت کردیم. گفتیم توی سی شارپ فانکشن های متنوعی داریم مثل: «Methods, properties, indexers, and events»

همه این فانکشن ها اگر قبلشون کلمه کلیدی `virtual` قرار بگیره میتونند توی `subclass` های خودشون `override` بشن.

خب بیایم یه پراپرتی به کلاس `Asset` اضافه کنیم.

حالا `subclass` های این کلاس اگه دلشون بخواد میتونند

پراپرتی رو `override` کنند و مقدار دلخواهشون رو

برگردونند. در غیر اینصورت مقدار پیش فرض یعنی صفر

برگردانده میشود.

```
6 references | 0 changes | 0 authors, 0 changes
public class Asset
{
    public string Name;
    1 reference | 0 changes | 0 authors, 0 changes
    public virtual decimal Liability => 0;    // Expression-bodied property
}

17 references | 0 changes | 0 authors, 0 changes
public class Stock : Asset    // inherits from Asset
{
    public long SharesOwned;
}

7 references | 0 changes | 0 authors, 0 changes
public class House : Asset    // inherits from Asset
{
    public decimal Mortgage;
    1 reference | 0 changes | 0 authors, 0 changes
    public override decimal Liability => Mortgage;
}
```

Virtual Function Members

```

8 references | 0 changes | 0 authors, 0 changes
public class Asset
{
    public string Name;
    4 references | 0 changes | 0 authors, 0 changes
    public virtual decimal Liability => 0; // Expression-bodied property
    2 references | 0 changes | 0 authors, 0 changes
    public virtual Asset Clone() => new Asset { Name = Name };
}

17 references | 0 changes | 0 authors, 0 changes
public class Stock : Asset // inherits from Asset
{
    public long SharesOwned;
    1 reference | 0 changes | 0 authors, 0 changes
    public override Stock Clone() => new Stock
    {
        Name = Name,
        SharesOwned = SharesOwned,
    };
}

10 references | 0 changes | 0 authors, 0 changes
public class House : Asset // inherits from Asset
{
    public decimal Mortgage;
    4 references | 0 changes | 0 authors, 0 changes
    public override decimal Liability => Mortgage;
    1 reference | 0 changes | 0 authors, 0 changes
    public override House Clone() => new House
    {
        Name = Name,
        Mortgage = Mortgage
    };
}

```

به این مثال دقت کنید:

کلاس **Asset** یک متد **virtual** برای خودش دارد که خروجیش یه آبجکت جدید از جنس خودشه.

حالا کلاسهای مشتق شده ازش اومدن و این متد رو **override** کردن و خروجی رو هم تغییر دادند ولی چرا کار میکنه؟

به این خاطر که متد در کلاس والد قرار بوده نوع **Asset** برگردونه و همونطور که گفتیم **subclass**ها میتونند جای کلاس والدشون بشینند پس **stock** و **House** هم یه جورایی همون **Asset** هستند و در نتیجه کد بدون مشکل اجرا میشه.

البته این امکان از **C#9** به بعد اضافه شده و قبل اون می بایست نوع خروجی حتما با نوع خروجی اصلی یکی بود.

Abstract Classes and Abstract Members

وقتی یه کلاس رو به صورت **abstract** تعریف میکنیم نمیتونیم ازش نمونه بسازیم، به عنوان مثال اگر کلاس **Asset** رو **abstract** کنیم متد **clone** به خطا میخوره.
کلاس های **abstract** میتونند اعضای **abstract** هم داشته باشند.

```
8 references | 0 changes | 0 authors, 0 changes
public abstract class Asset
{
    public string Name;
    4 references | 0 changes | 0 authors, 0 changes
    public virtual decimal Liability => 0;    // Expression-bodied property
    2 references | 0 changes | 0 authors, 0 changes
    public abstract decimal NetValue { get; }
    2 references | 0 changes | 0 authors, 0 changes
    public virtual Asset Clone() => new Asset { Name = Name };
}
```

```
17 references | 0 changes | 0 authors, 0 changes
public class Stock : Asset // inherits
{
    public long SharesOwned;
}
```

```
1 reference | 0 changes | 0 authors, 0 changes
```

💡 Asset.Asset()

CS0144: Cannot create an instance of the abstract type or interface 'Asset'

IDE0090: 'new' expression can be simplified

Show potential fixes (Ctrl+.)

Abstract Classes and Abstract Members

```
public abstract class Asset
{
    public string Name;
    4 references | 0 changes | 0 authors, 0 changes
    public virtual decimal Liability => 0;    // Expression-bodied property
    2 references | 0 changes | 0 authors, 0 changes
    public abstract decimal NetValue { get; }
    2 references | 0 changes | 0 authors, 0 changes
    public abstract Asset Clone();
}

17 references | 0 changes | 0 authors, 0 changes
public class Stock : Asset    // inherits from Asset
{
    public long SharesOwned;
    public decimal CurrentPrice;

    1 reference | 0 changes | 0 authors, 0 changes
    public override decimal NetValue => CurrentPrice * SharesOwned;

    1 reference | 0 changes | 0 authors, 0 changes
    public override Stock Clone()...
}

10 references | 0 changes | 0 authors, 0 changes
public class House : Asset    // inherits from Asset
{
    public decimal Mortgage;
    4 references | 0 changes | 0 authors, 0 changes
    public override decimal Liability => Mortgage;

    1 reference | 0 changes | 0 authors, 0 changes
    public override decimal NetValue => throw new NotImplementedException();

    1 reference | 0 changes | 0 authors, 0 changes
    public override House Clone()...
}
```

اعضای **abstract** مثل اعضای **virtual** می‌موند با این تفاوت که پیاده سازی پیش فرض نمیتونند داشته باشند و حتما توسط subclassها باید پیاده سازی شوند. (برعکس **virtual**ها)

با این اوصاف برای رفع خطایی که در اسلاید قبل اشاره شد میتونیم متد **clone** رو به صورت **abstract** تعریف کنیم.

Hiding Inherited Members

کلاس والد و کلاس مشتق شده از اون میتونند پراپرتی های هم نام داشته باشند. مثال زیر رو ببینید:

اتفاقی که میفته در کلاس B متغیر **counter** متغیر مشابه خود در کلاس والد رو از بین میبره و عملاً دیگه از کلاس B به پراپرتی **counter** در کلاس والد دسترسی نداریم.

```
Console.WriteLine(new A().Counter);  
Console.WriteLine(new B().Counter);
```

1 reference | 0 changes | 0 authors, 0 changes

```
public class A { public int Counter = 1; }
```

0 references | 0 changes | 0 authors, 0 changes

```
public class B : A { public int Counter = 2; }
```



(field) int B.Counter

CS0108: 'B.Counter' hides inherited member 'A.Counter'. Use the new keyword if hiding was intended.

Show potential fixes (Ctrl+.)

Hiding Inherited Members

```

4 references | 0 changes | 0 authors, 0 changes
public class BaseClass
{
    4 references | 0 changes | 0 authors, 0 changes
    public virtual void Foo() { Console.WriteLine("BaseClass.Foo"); }
}

2 references | 0 changes | 0 authors, 0 changes
public class Override : BaseClass
{
    4 references | 0 changes | 0 authors, 0 changes
    public override void Foo() { Console.WriteLine("Override.Foo"); }
}

2 references | 0 changes | 0 authors, 0 changes
public class Hider : BaseClass
{
    1 reference | 0 changes | 0 authors, 0 changes
    public new void Foo() { Console.WriteLine("Hider.Foo"); }
}

```

```

Override over = new Override();
BaseClass b1 = over;
over.Foo();           // Override.Foo
b1.Foo();             // Override.Foo

Hider hider = new Hider();
BaseClass b2 = hider;
hider.Foo();          // Hider.Foo
b2.Foo();             // BaseClass.Foo

```

این اتفاق معمولاً به ندرت می‌فته ولی اگر حالا خواستید واقعاً به متغیر رو عمداً **hide** کنید میتونید از کلمه کلیدی **new** استفاده کنید. این کار برای اینیه که به کامپایلر و بقیه برنامه نویسا اطلاع بدید که این کار اتفاقی نبوده و عمداً انجام شده.

علاوه بر این میتونید از کلمه کلیدی **override** هم استفاده کنید، تفاوت این دوتا رو میتونید تو این کد ببینید:

Sealing Functions and Classes

یک عضو `override` شده کلاس رو میتونیم به صورت `sealed` تعریف کنیم تا جلوی `override` اون رو در `subclass` های اون بگیریم. برای مثال پراپرتی `Liability` در کلاس `house` رو به صورت `sealed` تعریف کردیم. حالا دیگه کلاسی که از کلاس `house` ارث بری کنه نمیتونه اون پراپرتی رو `override` کنه.

5 references | 0 changes | 0 authors, 0 changes

```
public sealed override decimal Liability => Mortgage;
```

0 references | 0 changes | 0 authors, 0 changes

```
public class Villa : House
```

```
{
```

6 references | 0 changes | 0 authors, 0 changes

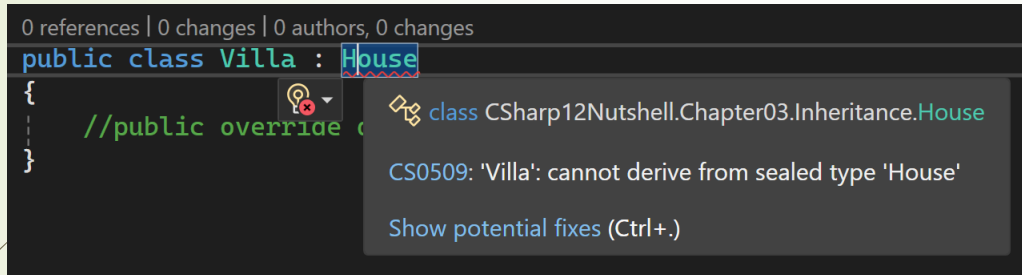
```
public override decimal Liability => base.Liability * 1.5M;
```

```
}
```

 decimal Villa.Liability { get; }

CS0239: 'Villa.Liability': cannot override inherited member 'House.Liability' because it is sealed

Sealing Functions and Classes



علاوه بر **function** ها، همیشه خود کلاس رو هم به صورت **Sealed** تعریف کرد. با این کار اجازه نمیدیم تا کلاسی از اون کلاس ارث بیره.

معمولا این کار نسبت به **sealed** کردن فانکشن ها رایج تره. البته اگرچه میتونید از **override** شدن اعضای یک کلاس جلوگیری کنید ولی از **hide** شدنشون همیشه جلوگیری کرد.

The base Keyword

کلمه کلیدی **base** خیلی شبیه کلمه کلیدی **this** هست و دو هدف زیر رو دنبال میکنه (ترجیح دادم متن اصلی و بیارم):

- Accessing an overridden function member from the subclass
- Calling a base-class constructor

5 references | 0 changes | 0 authors, 0 changes

```
public sealed override decimal Liability => base.Liability + Mortgage;
```

مثل این:

Constructors and Inheritance

Subclass ها میتوانند خودشون constructor داشته باشند constructor کلاس والد در کلاس مشتق شده در دسترس است اما به صورت اتوماتیک ارث برده نمیشه. Subclass هر کدوم از constructor ها رو که نیاز داشته باشه باید مجدد دوباره خودش تعریف کنه و با استفاده از کلمه کلیدی base به constructor والد دسترسی پیدا کنه.

```
4 references | 0 changes | 0 authors, 0 changes
public class Baseclass
{
    public int X;
    0 references | 0 changes | 0 authors, 0 changes
    public Baseclass() { }
    1 reference | 0 changes | 0 authors, 0 changes
    public Baseclass(int x) => X = x;
}

3 references | 0 changes | 0 authors, 0 changes
public class Subclass : Baseclass
{
    1 reference | 0 changes | 0 authors, 0 changes
    public Subclass(int x) : base(x) { }
}
```

کلمه کلیدی base مثل کلمه کلیدی this عمل میکنه با این تفاوت که constructor رو در کلاس پایه اجرا میکنه.

Constructor کلاس پایه همیشه اول اجرا میشه، این تضمین میکنه که اول کلاس پایه ساخته بشه بعد constructor مخصوص کلاس مشتق شده اجرا بشه.