Inheritance

A class can *inherit* from another class to extend or customize the original class. Inheriting from a class lets you reuse the functionality in that class instead of building it from scratch. A class can inherit from only a single class, but can itself be inherited by many classes, thus forming a class hierarchy. In this example, we start by defining a class called `Asset`:

```
public class Asset
{
  public string Name;
}
```

Next, we define classes called `Stock` and `House`, which will inherit from `Asset`. `Stock` and `House` get everything an `Asset` has, plus any additional members that they define:

```
public class Stock : Asset   // inherits from Asset
{
  public long SharesOwned;
}

public class House : Asset   // inherits from Asset
{
  public decimal Mortgage;
}
```

Here's how we can use these classes:

```
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
```

The *derived classes*, `Stock` and `House`, inherit the `Name` property from the *base class*, `Asset`.

Polymorphism

References are polymorphic. This means a variable of type *x* can refer to an object that subclasses *x*. For instance, consider the following method:

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

This method can display both a `Stock` and a `House`, since they are both `Assets`:

```
Stock msft    = new Stock ... ;
House mansion = new House ... ;

Display (msft);
Display (mansion);
```

Polymorphism works on the basis that subclasses (`Stock` and `House`) have all the features of their base class (`Asset`). The converse, however, is not true. If `Display` was modified to accept a `House`, you could not pass in an `Asset`:

```
static void Main() { Display (new Asset()); }    //
Compile-time error

public static void Display (House house)         //
Will not accept Asset
{
  System.Console.WriteLine (house.Mortgage);
}
```

Casting and Reference Conversions

An object reference can be:

- Implicitly *upcast* to a base class reference
- Explicitly *downcast* to a subclass reference

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. For example:

```
Stock msft = new Stock();

Asset a = msft;                  // Upcast
```

After the upcast, variable `a` still references the same `Stock` object as variable `msft`. The object being referenced is not itself altered or converted:

```
Console.WriteLine (a == msft);        // True
```

Although `a` and `msft` refer to the identical object, `a` has a more restrictive view on that object:

```
Console.WriteLine (a.Name);           // OK
Console.WriteLine (a.SharesOwned);    // Error:
SharesOwned undefined
```

The last line generates a compile-time error because the variable `a` is of type `Asset`, even though it refers to an object of type `Stock`. To get to its `SharesOwned` field, you must *downcast* the `Asset` to a `Stock`.

Downcasting

A downcast operation creates a subclass reference from a base class reference. For example:

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

As with an upcast, only references are affected—not the underlying object. A downcast requires an explicit cast because it can potentially fail at runtime:

```
House h = new House();
Asset a = h;                    // Upcast always succeeds
Stock s = (Stock)a;  // Downcast fails: a is not a stock
```

If a downcast fails, an `InvalidCastException` is thrown. This is an example of *runtime type checking* (we will elaborate on this concept in “Static and Runtime Type Checking”).

The as operator

The `as` operator performs a downcast that evaluates to `null` (rather than throwing an exception) if the downcast fails:

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

This is useful when you’re going to subsequently test whether the result is `null`:

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

NOTE

Without such a test, a cast is advantageous, because if it fails, a more helpful exception is thrown. We can illustrate by comparing the following two lines of code:

```
int shares = ((Stock)a).SharesOwned;    // Approach #1
int shares = (a as Stock).SharesOwned;  // Approach #2
```

If `a` is not a `Stock`, the first line throws an `InvalidCastException`, which is an accurate description of what went wrong. The second line throws a `NullReferenceException`, which is ambiguous. Was `a` not a `Stock` or was `a` null?

Another way of looking at it is that with the cast operator, you’re saying to the compiler: “I’m *certain* of a value’s type; if I’m wrong, there’s a bug in my code, so throw an exception!” Whereas with the `as` operator, you’re uncertain of its type and want to branch according to the outcome at runtime.

The is operator

The `is` operator tests whether a reference conversion would succeed; in other words, whether an object derives from a specified class (or implements an interface). It is often used to test before downcasting.

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

The `is` operator also evaluates to true if an *unboxing conversion* would succeed (see "The object Type"). However, it does not consider custom or numeric conversions.

The is operator and pattern variables (C# 7)

From C# 7, you can introduce a variable while using the `is` operator:

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

This is equivalent to:

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

The variable that you introduce is available for "immediate" consumption, so the following is legal:

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

Virtual Function Members

A function marked as `virtual` can be *overridden* by subclasses wanting to provide a specialized implementation. Methods, properties, indexers, and events can all be declared `virtual`:

```
public class Asset
{
  public string Name;
  public virtual decimal Liability => 0;    // Expression-
bodied property
}
Liability => 0 is a shortcut for { get { return 0; } }
```

A subclass overrides a virtual method by applying the `override` modifier:

```
public class Stock : Asset
{
  public long SharesOwned;
}
```

```
public class House : Asset
{
  public decimal Mortgage;
  public override decimal Liability => Mortgage;
}
```

By default, the `Liability` of an `Asset` is `0`. A `Stock` does not need to specialize this behavior. However, the `House`specializes the `Liability` property to return the value of the `Mortgage`:

```
House mansion = new House { Name="McMansion",
Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability);  // 250000
Console.WriteLine (a.Liability);        // 250000
```

The signatures, return types, and accessibility of the virtual and overridden methods must be identical. An overridden method can call its base class implementation via the `base` keyword.

Abstract Classes and Abstract Members

A class declared as *abstract* can never be instantiated. Instead, only its concrete *subclasses* can be instantiated.

Abstract classes are able to define *abstract members*. Abstract members are like virtual members, except they don't provide a default implementation. That implementation must be provided by the subclass, unless that subclass is also declared abstract:

```
public abstract class Asset
{
  // Note empty implementation
  public abstract decimal NetValue { get; }
}

public class Stock : Asset
{
  public long SharesOwned;
  public decimal CurrentPrice;

  // Override like a virtual method.
```

```
  public override decimal NetValue => CurrentPrice *
SharesOwned;
}
```

Hiding Inherited Members

A base class and a subclass may define identical members. For example:

```
public class A      { public int Counter = 1; }
public class B : A  { public int Counter = 2; }
```

The `Counter` field in class `B` is said to *hide* the `Counter` field in class `A`. Usually, this happens by accident, when a member is added to the base type *after* an identical member was added to the subtype. For this reason, the compiler generates a warning, and then resolves the ambiguity as follows:

- References to `A` (at compile time) bind to `A.Counter`.
- References to `B` (at compile time) bind to `B.Counter`.

Occasionally, you want to hide a member deliberately, in which case you can apply the `new` modifier to the member in the subclass.
The `new` modifier *does nothing more than suppress the compiler warning that would otherwise result*:

```
public class A     { public     int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

The `new` modifier communicates your intent to the compiler—and other programmers—that the duplicate member is not an accident.

new versus override

Consider the following class hierarchy:

```
public class BaseClass
{
  public virtual void Foo()  { Console.WriteLine
("BaseClass.Foo"); }
}

public class Overrider : BaseClass
{
  public override void Foo() { Console.WriteLine
("Overrider.Foo"); }
}
```

```
public class Hider : BaseClass
{
  public new void Foo()      { Console.WriteLine
("Hider.Foo"); }
}
```

The differences in behavior between `Overrider` and `Hider` are demonstrated in the following code:

```
Overrider over = new Overrider();
BaseClass b1 = over;
over.Foo();                          // Overrider.Foo
b1.Foo();                            // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();                            // Hider.Foo
b2.Foo();                           // BaseClass.Foo
```

Sealing Functions and Classes

An overridden function member may *seal* its implementation with the `sealed` keyword to prevent it from being overridden by further subclasses. In our earlier virtual function member example, we could have sealed `House`'s implementation of `Liability`, preventing a class that derives from `House` from overriding `Liability`, as follows:

```
public sealed override decimal Liability { get { return
Mortgage; } }
```

You can also seal the class itself, implicitly sealing all the virtual functions, by applying the `sealed` modifier to the class itself. Sealing a class is more common than sealing a function member.

Although you can seal against overriding, you can't seal a member against being *hidden*.

The base Keyword

The `base` keyword is similar to the `this` keyword. It serves two essential purposes:

- Accessing an overridden function member from the subclass

- Calling a base-class constructor (see the next section)

In this example, `House` uses the `base` keyword to access `Asset`'s implementation of `Liability`:

```
public class House : Asset
{
  ...
  public override decimal Liability => base.Liability +
Mortgage;
}
```

With the `base` keyword, we access `Asset`'s `Liability` property *nonvirtually*. This means we will always access `Asset`'s version of this property—regardless of the instance's actual runtime type.

The same approach works if `Liability` is *hidden* rather than *overridden*. (You can also access hidden members by casting to the base class before invoking the function.)

Constructors and Inheritance

A subclass must declare its own constructors. The base class's constructors are *accessible* to the derived class, but are never automatically *inherited*. For example, if we define `Baseclass` and `Subclass` as follows:

```
public class Baseclass
{
  public int X;
  public Baseclass () { }
  public Baseclass (int x) { this.X = x; }
}

public class Subclass : Baseclass { }
```

the following is illegal:

```
Subclass s = new Subclass (123);
```

`Subclass` must hence "redefine" any constructors it wants to expose. In doing so, however, it can call any of the base class's constructors with the `base` keyword:

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { }
}
```

The `base` keyword works rather like the `this` keyword, except that it calls a constructor in the base class.

Base-class constructors always execute first; this ensures that *base* initialization occurs before *specialized* initialization.

Implicit calling of the parameterless base-class constructor

If a constructor in a subclass 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); }  // 1
}
```

If the base class has no accessible parameterless constructor, subclasses are forced to use the `base` keyword in their constructors.

Constructor and field initialization order

When an object is instantiated, initialization takes place in the following order:

1. From subclass to base class:
   a. Fields are initialized.
   b. Arguments to base-class constructor calls are evaluated.
2. From base class to subclass:
   a. Constructor bodies execute.

The following code demonstrates:

```
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
  {
     ...               // Executes 5th
  }
```

}

Overloading and Resolution

Inheritance has an interesting impact on method overloading. Consider the following two overloads:

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

When an overload is called, the most specific type has precedence:

```
House h = new House (...);
Foo(h);                          // Calls Foo(House)
```

The particular overload to call is determined statically (at compile time) rather than at runtime. The following code calls `Foo(Asset)`, even though the runtime type of `a` is `House`:

```
Asset a = new House (...);
Foo(a);                          // Calls Foo(Asset)
```

NOTE

If you cast `Asset` to `dynamic`, the decision as to which overload to call is deferred until runtime, and is then based on the object's actual type:

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

Access Modifiers

To promote encapsulation, a type or type member may limit its *accessibility* to other types and other assemblies by adding one of five *access modifiers* to the declaration:

`public`

> Fully accessible. This is the implicit accessibility for members of an enum or interface.

`internal`

> Accessible only within the containing assembly

`private`

> Accessible only within the containing type. This is the default accessibility for members of a class or struct.

`protected`

> Accessible only within the containing type or subclasses.

`protected internal`

> The *union* of `protected` and `internal` accessibility. Eric Lippert explains it as follows: Everything is as private as possible by default, and each modifier makes the thing *more accessible*. So something that is `protected internal` is made more accessible in two ways.

Examples

`Class2` is accessible from outside its assembly; `Class1` is not:

```
class Class1 {}                  // Class1 is
internal (default)

public class Class2 {}
```

`ClassB` exposes field `x` to other types in the same assembly; `ClassA` does not:

```
class ClassA { int x;          } // x is private
(default)

class ClassB { internal int x; }
```

Functions within `Subclass` can call `Bar` but not `Foo`:

```
class BaseClass
{
  void Foo()           {}        // Foo is
private (default)
  protected void Bar() {}
}

class Subclass : BaseClass
{
  void Test1() { Foo(); }       // Error -
cannot access Foo
  void Test2() { Bar(); }       // OK
}
```