```
}
```

The application cannot compile, because `Widget` is ambiguous. Extern aliases can resolve the ambiguity in our application:

```
// csc /r:W1=Widgets1.dll /r:W2=Widgets2.dll application.cs
```

```
extern alias W1;
extern alias W2;
class Test
{
  static void Main()
  {
    W1.Widgets.Widget w1 = new W1.Widgets.Widget();
    W2.Widgets.Widget w2 = new W2.Widgets.Widget();
  }
}
```

Namespace alias qualifiers

As we mentioned earlier, names in inner namespaces hide names in outer namespaces. However, sometimes even the use of a fully qualified type name does not resolve the conflict. Consider the following example:

```
namespace N
```

```
namespace N
{
  class A
  {
    public class B {}
    static void Main() { new A.B(); }
  }
}

// Nested type
// Instantiate class B

namespace A
{
```

```
  namespace A
  {
    class B {}
  }
```

The `Main` method could be instantiating either the nested class `B`, or the class `B` within the namespace `A`. The compiler always gives higher precedence to identifiers in the current namespace; in this case, the nested `B` class.

To resolve such conflicts, a namespace name can be qualified, relative to one of the following:

-
-

The global namespace—the root of all namespaces (identified with the contextual keyword `global`)

The set of extern aliases

The :: token is used for namespace alias qualification. In this example, we qualify using the global namespace (this is most commonly seen in auto-generated code to avoid name conflicts):

```
namespace N
{
  class A
  {
    static void Main()
    {
      System.Console.WriteLine (new A.B());
      System.Console.WriteLine (new global::A.B());
    }
    public class B {}
  }
}
```

```
  }
}
namespace A
{
  class B {}
}
```

Here is an example of qualifying with an alias (adapted from the example in "Extern" on page 60):

```
extern alias W1;
extern alias W2;

class Test
{
  static void Main()
  {
    W1::Widgets.Widget w1 = new W1::Widgets.Widget();
```

```
    {
      W1::Widgets.Widget w1 = new W1::Widgets.Widget();
      W2::Widgets.Widget w2 = new W2::Widgets.Widget();
    }
}
```


3

Creating Types in C#

Creating Types in C#

In this chapter, we will delve into types and type members.

Classes

A class is the most common kind of reference type. The simplest possible class declaration is as follows:

```
class YourClassName
{
}
```

A more complex class optionally has the following:

Preceding the keyword `class`

Following `YourClassName`

Following `YourClassName`

Within the braces

Attributes and *class modifiers*. The non-nested class modifiers are `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe`, and `partial`

Generic type parameters, a *base class*, and *interfaces*

Class members (these are *methods*, *properties*, *indexers*, *events*, *fields*, *constructors*, *operator functions*, *nested types*, and a *finalizer*)

This chapter covers all of these constructs except attributes, operator functions, and the `unsafe` keyword, which are covered in Chapter 4. The following sections will enumerate each of the class members.

Fields

A *field* is a variable that is a member of a class or struct. For example:

A *field* is a variable that is a member of a class or struct. For example:

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

63

Fields allow the following modifiers:

Static modifier
Access modifiers

Access modifiers

Inheritance modifier

Unsafe code modifier

Read-only modifier

Threading modifier

`static`

`public internal private protected`

`new`

`unsafe`

`readonly`

`volatile`

The readonly modifier

The `readonly` modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or within the enclosing type's constructor.

Field initialization

Field initialization is optional. An uninitialized field has a default value (`0`, `\0`, `null`, `false`). Field initializers run before constructors:

```
public int Age = 10;
```

Declaring multiple fields together

For convenience, you may declare multiple fields of the same type in a comma-separated list. This is a convenient way for all the fields to share the same attributes

separated list. This is a convenient way for all the fields to share the same attributes and field modifiers. For example:

```
static readonly int legs = 8,
                    eyes = 1;
```

Methods

A method performs an action in a series of statements. A method can receive *input* data from the caller by specifying *parameters* and *output* data back to the caller by specifying a *return type*. A method can specify a `void` return type, indicating that it doesn't return any value to its caller. A method can also output data back to the caller via `ref`/`out` parameters.

A method's *signature* must be unique within the type. A method's signature comprises its name and parameter types (but not the parameter *names*, nor the return type).

Methods allow the following modifiers:

Methods allow the following modifiers:

- Static modifier
- Access modifiers
- Inheritance modifiers
- Unmanaged code modifiers

- `static`
- `public internal private protected`
- `new virtual abstract override sealed`
- `unsafe extern`

Overloading methods

Overloading methods

A type may overload methods (have multiple methods with the same name), as long as the signatures are different. For example, the following methods can all coexist in the same type:

```
void Foo (int x);
void Foo (double x);
void Foo (int x, float y);
void Foo (float x, int y);
```

However, the following pairs of methods cannot coexist in the same type, since the return type and the `params` modifier are not part of a method's signature:

```
void  Foo (int x);
float Foo (int x);           // Compile-time error
```

```
void
void

Goo (int[] x);
```

```
Goo (int[] x);
Goo (params int[] x);

// Compile-time error
```

Pass-by-value versus pass-by-reference

Whether a parameter is pass-by-value or pass-by-reference is also part of the signature. For example, `Foo(int)` can coexist with either `Foo(ref int)` or `Foo(out int)`. However, `Foo(ref int)` and `Foo(out int)` cannot coexist:

```
void Foo (int x);
void Foo (ref int x);      // OK so far
void Foo (out int x);      // Compile-time error
```

Instance Constructors

Constructors run initialization code on a class or struct. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class Panda
{
  string name;
```

```
{
   string name;
   public Panda (string n)
   {
      name = n;
   }
}

// Define field
// Define constructor
// Initialization code (set up field)
// Call constructor
...
```

```
...
Panda p = new Panda ("Petey");
```

Constructors allow the following modifiers:

Access modifiers
Unmanaged code modifiers

```
public internal private protected
unsafe extern
```


Overloading constructors

Overloading constructors

A class or struct may overload constructors. To avoid code duplication, one constructor may call another, using the `this` keyword:

```
using System;

public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

When one constructor calls another, the *called constructor* executes first.

You can pass an *expression* into another constructor as follows:

```
public Wine (decimal price, DateTime year) : this (price, year.Year) { }
```

The expression itself cannot make use of the `this` reference, for example, to call an instance method. It can, however, call static methods.

instance method. It can, however, call static methods.

Implicit parameterless constructors

For classes, the C# compiler automatically generates a parameterless constructor if and only if you do not define any constructors. However, as soon as you define at least one constructor, the parameterless constructor is no longer automatically generated.

For structs, a parameterless constructor is intrinsic to the struct; therefore, you cannot define your own. The role of a struct's implicit parameterless constructor is to initialize each field with default values.

Constructor and field initialization order

Previously, we saw that fields can be initialized with default values in their declaration:

```
class Player
{
```

```
{
  int shields = 50;
  int health = 100;
}

// Initialized first
// Initialized second
```

Field initializations occur *before* the constructor is executed, and in the declaration order of the fields.

Nonpublic constructors

Constructors do not need to be public. A common reason to have a nonpublic constructor is to control instance creation via a static method call. The static method could be used to return an object from a pool rather than necessarily creating a new object, or return various subclasses based on input arguments. The template for that

could be used to return an object from a pool rather than necessarily creating a new object, or return various subclasses based on input arguments. The template for that pattern is shown next:


```
public class Class1
{
  Class1() {}                                    // Private constructor
  public static Class1 Create (...)
  {
    // Perform custom logic here to return an instance of Class1
    ...
  }
}
```

Object Initializers

To simplify object initialization, the accessible fields or properties of an object can be initialized in a single statement directly after construction. For example, consider

To simplify object initialization, the accessible fields or properties of an object can be initialized in a single statement directly after construction. For example, consider the following class:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;
```

```
    public Bunny () {}
    public Bunny (string n) { Name = n; }
}
```

Using object initializers, you can instantiate `Bunny` objects as follows:

```
// Note parameterless constructors can omit empty parentheses
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo")     { LikesCarrots=true, LikesHumans=false };
```

The code to construct `b1` and `b2` is precisely equivalent to:

```
Bunny temp1 = new Bunny();    // temp1 is a compiler-generated name
temp1.Name = "Bo";
temp1.LikesCarrots = true;
```

```
temp1.Name = "Bo";
temp1.LikesCarrots = true;
temp1.LikesHumans = false;
Bunny b1 = temp1;
```

```
Bunny temp2 = new Bunny ("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

The temporary variables are to ensure that if an exception is thrown during initialization, you can't end up with a half-initialized object.

Object initializers were introduced in C# 3.0.

Object Initializers Versus Optional Parameters

Object Initializers Versus Optional Parameters

Instead of using object initializers, we could make `Bunny`'s constructor accept optional parameters:

```
public Bunny (string name,
              bool likesCarrots = false,
              bool likesHumans = false)
{
  Name = name;
  LikesCarrots = likesCarrots;
  LikesHumans = likesHumans;
}
```

This would allow us to construct a `Bunny` as follows:

```
Bunny b1 = new Bunny (name: "Bo",
                      likesCarrots: true);
```

An advantage of this approach is that we could make `Bunny`'s fields (or *properties*, as we'll explain shortly) read-only if we choose. Making fields or properties read-only is good practice when there's no valid reason for them to change throughout the life of the object.

The disadvantage in this approach is that each optional parameter value is baked into the *calling site*. In other words, C# translates our constructor call into this:

into the *calling site*. In other words, C# translates our constructor call into this:

```
Bunny b1 = new Bunny ("Bo", true, false);
```

This can be problematic if we instantiate the `Bunny` class from another assembly, and later modify `Bunny` by adding another optional parameter—such as `likesCats`. Unless the referencing assembly is also recompiled, it will continue to call the (now nonexistent) constructor with three parameters and fail at runtime. (A subtler problem is that if we changed the value of one of the optional parameters, callers in other assemblies would continue to use the old optional value until they were recompiled.)

Hence, optional parameters are best avoided in public functions if you want to offer binary compatibility between assembly versions.

The this Reference

The `this` reference refers to the instance itself. In the following example, the `Marry` method uses `this` to set the `partner`'s `mate` field:

```
public class Panda
```

```
public class Panda
{
  public Panda Mate;

  public void Marry (Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```


The this reference also disambiguates a local variable or parameter from a field. For example:

The this reference also disambiguates a local variable or parameter from a field. For example:

```
public class Test
{
  string name;
  public Test (string name) { this.name = name; }
}
```

The this reference is valid only within nonstatic members of a class or struct.

Properties

Properties look like fields from the outside, but internally they contain logic, like methods do. For example, you can't tell by looking at the following code whether CurrentPrice is a field or a property:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

A property is declared like a field, but with a `get`/`set` block added. Here's how to implement `CurrentPrice` as a property:

```
public class Stock
{
  decimal currentPrice;
```

```
    decimal currentPrice;

  // The private "backing" field

  public decimal CurrentPrice     // The public property
  {
    get { return currentPrice; } set { currentPrice = value; }
  }
}
```

get and set denote property *accessors*. The get accessor runs when the property is read. It must return a value of the property's type. The set accessor is run when the property is assigned. It has an implicit parameter named value of the property's type that you typically assign to a private field (in this case, currentPrice).

Although properties are accessed in the same way as fields, they differ in that they give the implementer complete control over getting and setting its value. This control enables the implementer to choose whatever internal representation is needed, without exposing the internal details to the user of the property. In this example, the set method could throw an exception if value was outside a valid range of values.

the set method could throw an exception if value was outside a valid range of values.

Throughout this book, we use public fields extensively to keep the examples free of distraction. In a real application, you would typically favor public properties over public fields, in order to promote encapsulation.

Properties allow the following modifiers:

- Static modifier
- Access modifiers
- Inheritance modifiers
- Unmanaged code modifiers

```
static
public internal private protected
new virtual abstract override sealed
unsafe extern
```

Read-only and calculated properties

A property is read-only if it specifies only a **get** accessor, and it is write-only if it specifies only a **set** accessor. Write-only properties are rarely used.

A property typically has a dedicated backing field to store the underlying data. However, a property can also be computed from other data. For example:

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
```

```
{
  get { return currentPrice * sharesOwned; }
}
```

Automatic properties

The most common implementation for a property is a getter and/or setter that simply reads and writes to a private field of the same type as the property. An *automatic property* declaration instructs the compiler to provide this implementation. We can redeclare the first example in this section as follows:

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set; }
}
```

The compiler automatically generates a private backing field of a compiler-generated

The compiler automatically generates a private backing field of a compiler-generated name that cannot be referred to. The `set` accessor can be marked `private` if you want to expose the property as read-only to other types. Automatic properties were introduced in C# 3.0.

get and set accessibility

The `get` and `set` accessors can have different access levels. The typical use case for this is to have a `public` property with an `internal` or `private` access modifier on the setter:

```
public class Foo
{
  private decimal x;
  public decimal X
  {
    get         { return x;  }
    private set { x = Math.Round (value, 2); }
```

```
    private set { x = Math.Round (value, 2); }
```



```
    }
  }
```

Notice that you declare the property itself with the more permissive access level (`public`, in this case), and then add the modifier to the accessor that you want to be *less* accessible.

CLR property implementation

C# property accessors internally compile to methods called `get_XXX` and `set_XXX`:

```
public int  get_CurrentPrice {...}
public void set_CurrentPrice (decimal value) {...}
```

Simple nonvirtual property accessors are *inlined* by the JIT (Just-In-Time) compiler,

Simple nonvirtual property accessors are *inlined* by the JIT (Just-In-Time) compiler, eliminating any performance difference between accessing a property and a field. Inlining is an optimization in which a method call is replaced with the body of that method.

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties, but are accessed via an index argument rather than a property name. The `string` class has an indexer that lets you access each of its `char` values via an `int` index:

```
string s = "hello";
Console.WriteLine (s[0]); // 'h'
Console.WriteLine (s[3]); // 'l'
```

The syntax for using indexers is like that for using arrays when the index is an integer type.

Indexers have the same modifiers as properties (see "Properties" on page 69).

Implementing an indexer

To write an indexer, define a property called `this`, specifying the arguments in square brackets. For instance:

```
class Sentence
{
  string[] words = "The quick brown fox".Split();

  public string this [int wordNum]      // indexer
  {
    get { return words [wordNum];  }
    set { words [wordNum] = value; }
  }
}
```

Here's how we could use this indexer:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);
s[3] = "kangaroo";
Console.WriteLine (s[3]);

// fox
// kangaroo
```

A type may declare multiple indexers, each with parameters of different types. An indexer can also take more than one parameter:

```
public string this [int arg1, string arg2]
```

```
public string this [int arg1, string arg2]
{
  get { ... }  set { ... }
}
```

If you omit the **set** accessor, an indexer becomes read-only.

CLR indexer implementation

Indexers internally compile to methods called `get_Item` and `set_Item`, as follows:

```
public string get_Item (int wordNum) {...}
public void set_Item (int wordNum, string value) {...}
```

The compiler chooses the name "Item" by default—you can actually change this by decorating your indexer with the following attribute:

```
[System.Runtime.CompilerServices.IndexerName ("Blah")]
```

Constants

A *constant* is a field whose value can never change. A constant is evaluated statically at compile time and the compiler literally substitutes its value whenever used, rather

A *constant* is a field whose value can never change. A constant is evaluated statically at compile time and the compiler literally substitutes its value whenever used, rather like a macro in C++. A constant can be any of the built-in numeric types, `bool`, `char`, `string`, or an enum type.

A constant is declared with the `const` keyword and must be initialized with a value. For example:

```
public class Test
{
  public const string Message = "Hello World";
}
```

A constant is much more restrictive than a `static readonly` field—both in the types you can use and in field initialization semantics. A constant also differs from a `static readonly` field in that the evaluation of the constant occurs at compile time. For example:

```
public static double Circumference (double radius)
{
  return 2 * System.Math.PI * radius;
}
```

```
    return 2 * System.Math.PI * radius;
  }
```

is compiled to:


```
public static double Circumference (double radius)
{
  return 6.2831853071795862 * radius;
}
```

It makes sense for `PI` to be a constant, since it can never change. In contrast, a `static readonly` field can have a different value per application.

A `static readonly` field is also advantageous when exposing to other assemblies a value that might change in a later version. For instance, suppose assembly `X` exposes a constant as follows:

For instance, suppose assembly `X` exposes a constant as follows:

```
public const int MaximumThreads = 20;
```

If assembly `Y` references `X` and uses this constant, the value `20` will be baked into assembly `Y` when compiled. This means that if `X` is later recompiled with the constant set to 50, `Y` will still use the old value of 20 *until* `Y` *is recompiled*. A `static readonly` field avoids this problem.

Constants can also be declared local to a method. For example:

```
static void Main()
{
  const double twoPI  = 2 * System.Math.PI;
  ...
}
```

Constants allow the following modifiers:

- Access modifiers
- Inheritance modifier

`public internal private protected new`

new

Static Constructors

A static constructor executes once per *type*, rather than once per *instance*. A type can define only one static constructor, and it must be parameterless and have the same name as the type:

```
class Test
{
  static Test() { Console.WriteLine ("Type Initialized"); }
}
```

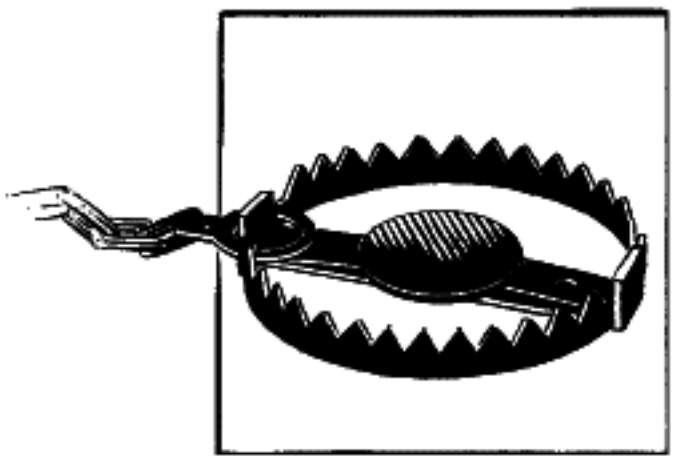
The runtime automatically invokes a static constructor just prior to the type being used. Two things trigger this:

-
-

Instantiating the type

Instantiating the type
Accessing a static member in the type

The only modifiers allowed by static constructors are `unsafe` and `extern`.



If a static constructor throws an unhandled exception (Chapter 4), that type becomes *unusable* for the life of the application.

Static constructors and field initialization order

Static constructors and field initialization order

Static field initializers run just *before* the static constructor is called. If a type has no static constructor, field initializers will execute just prior to the type being used—or *anytime earlier* at the whim of the runtime. (This means that the presence of a static constructor may cause field initializers to execute later in the program than they would otherwise.)

Static field initializers run in the order in which the fields are declared. The following example illustrates this: `X` is initialized to 0 and `Y` is initialized to 3.

```
class Foo
{
  public static int X = Y;
  public static int Y = 3;
}
```

```
// 0
// 3
```

If we swap the two field initializers around, both fields are initialized to 3. The next example prints 0 followed by 3 because the field initializer that instantiates a `Foo` executes before `X` is initialized to 3:

```
class Program
{
  static void Main() { Console.WriteLine (Foo.X); }
}

class Foo
{
  public static Foo Instance = new Foo();
  public static int X = 3;

  Foo() { Console.WriteLine (X); }   // 0
```

```
    Foo() { Console.WriteLine (X); }   // 0
}

// 3
```

If we swap the two lines in boldface, the example prints 3 followed by 3.

Static Classes

A class can be marked `static`, indicating that it must be composed solely of static members and cannot be subclassed. The `System.Console` and `System.Math` classes are good examples of static classes.

Finalizers

Finalizers are class-only methods that execute before the garbage collector reclaims the memory for an unreferenced object. The syntax for a finalizer is the name of the class prefixed with the ~ symbol:

class prefixed with the ~ symbol:


```
class Class1
{
  ~Class1()
  {
    ...
  }
}
```

This is actually C# syntax for overriding `Object`'s `Finalize` method, and the compiler expands it into the following method declaration:

```
protected override void Finalize()
{
```

```
protected override void Finalize()
{
  ...
  base.Finalize();
}
```

We discuss garbage collection and finalizers fully in Chapter 12.

Finalizers allow the following modifier:

Unmanaged code modifier

unsafe

Partial Types and Methods

Partial types allow a type definition to be split—typically across multiple files. A common scenario is for a partial class to be auto-generated from some other source (e.g., an XSD), and for that class to be augmented with additional hand-authored methods. For example:

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }
```

```
// PaymentForm.cs - hand-authored
partial class PaymentForm { ... }
```

```
partial class PaymentForm { ... }
```

Each participant must have the `partial` declaration; the following is illegal:

```
partial class PaymentForm {}
class PaymentForm {}
```

Participants cannot have conflicting members. A constructor with the same parameters, for instance, cannot be repeated. Partial types are resolved entirely by the compiler, which means that each participant must be available at compile time and must reside in the same assembly.

There are two ways to specify a base class with partial classes:

-
-

Specify the (same) base class on each participant. For example:

```
partial class PaymentForm : ModalForm {}
partial class PaymentForm : ModalForm {}
```

```
partial class PaymentForm : ModalForm {}
```

Specify the base class on just one participant. For example:

```
partial class PaymentForm : ModalForm {}
partial class PaymentForm {}
```


In addition, each participant can independently specify interfaces to implement. We cover base classes and interfaces in "Inheritance" on page 76 and "Interfaces" on page 92.

Partial methods

A partial type may contain *partial methods*. These let an auto-generated partial type provide customizable hooks for manual authoring. For example:

```
partial class PaymentForm    // In auto-generated file
{
```

```
    {
    ...
    partial void ValidatePayment (decimal amount);
}

partial class PaymentForm
{

// In hand-authored file
    ...
    partial void ValidatePayment (decimal amount)
    {
        if (amount > 100)
            ...
    }
}
```

```
    }
}
```

A partial method consists of two parts: a *definition* and an *implementation*. The definition is typically written by a code generator, and the implementation is typically manually authored. If an implementation is not provided, the definition of the partial method is compiled away. This allows auto-generated code to be liberal in providing hooks, without having to worry about code bloat. Partial methods must be `void` and are implicitly `private`.

Partial methods were introduced in C# 3.0.

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 class Asset
{
  public string Name;
}
```

Next, we define classes called `Stock` and `House`, which will inherit from `Asset`. They 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
{
```

```
{
  public decimal Mortgage;
}

// inherits from Asset
```

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);
Console.WriteLine (mansion.Mortgage);

// Mansion
// 250000
```

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

A subclass is also called a *derived class*.

A base class is also called a *superclass*.

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);
}
```

```
    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)
{
```

```
{
  System.Console.WriteLine (house.Mortgage);
}

// Will not accept Asset
```


Casting and Reference Conversions

An object reference can be:

-
-

Implicitly *upcast* to a base class reference

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 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:

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 SharedOwned 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
```

```
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" on page 87).

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);
```

Without such a test, a cast is advantageous, because if it fails, a more descriptive 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?

The as operator cannot perform *custom conversions* (see "Operator Overloading" on page 153 in Chapter 4) and it cannot do numeric conversions:

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

The as and cast operators will also perform upcasts, although this is not terribly useful because an implicit conversion will do the job.

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

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 does not consider custom or numeric conversions, but it does consider *unboxing conversions* (see "The object Type" on page 85).

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 string Name;
  public virtual decimal Liability { 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 { get { return Mortgage; } }
}
```

By default, the `Liability` of an `Asset` is 0. A `Stock` does not need to specialize this

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 };
Console.WriteLine (mansion.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 (we will cover this in "The base Keyword" on page 82).

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
```

```
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
{
  get { return CurrentPrice * SharesOwned; }
```

```
    get { return 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 programmers—that the duplicate member is not an accident.

other

C# overloads the new keyword to have independent meanings in different contexts. Specifically, the new *operator* is different from the new *member* modifier.

new versus virtual

new versus virtual

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 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();
b1.Foo();
Hider h = new Hider();
BaseClass b2 = h;
```

```
BaseClass b2 = h;
h.Foo();
b2.Foo();
```

```
 // Overrider.Foo
 // Overrider.Foo
// Hider.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

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.

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)

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
  {
    get { return 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. For example, if we define `Subclass` as follows:

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

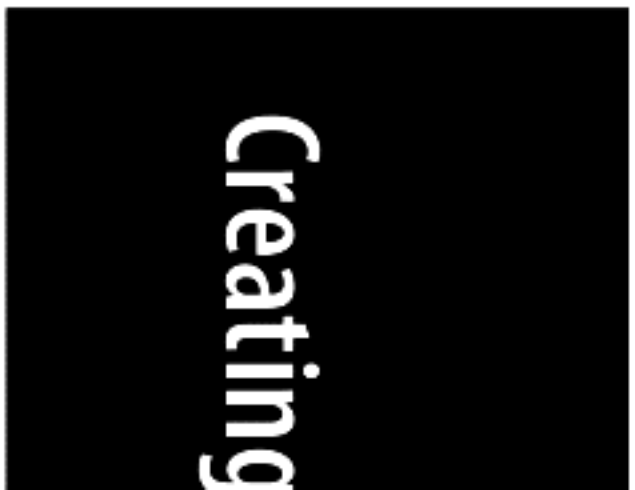
the following is illegal:

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 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 parameterless constructor, subclasses are forced to use the `base` keyword in their constructors.

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 = 0;
   public B (int x)
   {
      ...
   }
}
public class D : B
{
   int y = 0;
```

```
    int y = 0;
    public D (int x)
      : base (x + 1)
    {
        ...
    }
}

// Executes 3rd
// Executes 4th
// Executes 1st
// 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 (...);
```

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

// Calls Foo(Asset)
```


If you cast `Asset` to `dynamic` (Chapter 4), 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)
```

The object Type

`object` (`System.Object`) is the ultimate base class for all types. Any type can be upcast to `object`.

To illustrate how this is useful, consider a general-purpose *stack*. A stack is a data structure based on the principle of LIFO—"Last-In First-Out." A stack has two operations: *push* an object on the stack, and *pop* an object off the stack. Here is a simple implementation that can hold up to 10 objects:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)   { data[position++] = obj;  }
  public object Pop()             { return data[--position]; }
}
```

Because Stack works with the object type, we can Push and Pop instances of *any type* to and from the Stack:

```
Stack stack = new Stack();
stack.Push ("sausage");
string s = (string) stack.Pop();   // Downcast, so explicit cast is needed
```

```
Console.WriteLine (s);
```

```
// sausage
```

```
// Sausage
```

`object` is a reference type, by virtue of being a class. Despite this, value types, such as `int`, can also be cast to and from `object`, and so be added to our stack. This feature of C# is called *type unification* and is demonstrated here:

```
stack.Push (3);
int three = (int) stack.Pop();
```

When you cast between a value type and `object`, the CLR must perform some special work to bridge the difference in semantics between value and reference types. This process is called *boxing* and *unboxing*.

In "Generics" on page 101, we'll describe how to improve our `Stack` class to better handle stacks with same-typed elements.

Boxing and Unboxing

Boxing and Unboxing

Boxing is the act of casting a value-type instance to a reference-type instance. The reference type may be either the `object` class or an interface (which we will visit later in the chapter).* In this example, we box an `int` into an object:

```
int x = 9;
object obj = x;

// Box the int
```

Unboxing reverses the operation, by casting the object back to the original value type:

```
int y = (int)obj;          // Unbox the int
```

Unboxing requires an explicit cast. The runtime checks that the stated value type matches the actual object type, and throws an `InvalidCastException` if the check fails. For instance, the following throws an exception, because `long` does not exactly

matches the actual object type, and throws an `InvalidCastException` if the check fails. For instance, the following throws an exception, because `long` does not exactly match `int`:

```
object obj = 9;
long x = (long) obj;

// 9 is inferred to be of type int
// InvalidCastException
```

The following succeeds, however:

```
object obj = 9;
long x = (int) obj;
```

As does this:

As does this:

```
object obj = 3.5;
int x = (int) (double) obj;

// 3.5 is inferred to be of type double
// x is now 3
```

In the last example, `(double)` performs an *unboxing* and then `(int)` performs a *numeric conversion*.

Boxing conversions are crucial in providing a unified type system. The system is not perfect, however: we'll see in "Generics" on page 101 that variance with arrays and generics supports only *reference conversions* and not *boxing conversions*:

```
object[] a1 = new string[3];
```

```
object[] a1 = new string[3];
object[] a2 = new int[3];

// Legal
// Error
```

Copying semantics of boxing and unboxing

Boxing *copies* the value-type instance into the new object, and unboxing *copies* the contents of the object back into a value-type instance. In the following example, changing the value of `i` doesn't change its previously boxed copy:

```
int i = 3;
object boxed = i;
i = 5;
```

```
i = 5;
Console.WriteLine (boxed);

// 3
```

* The reference type may also be `System.ValueType` or `System.Enum` (Chapter 6).

Static and Runtime Type Checking

C# checks types both statically (at compile time) and at runtime.

Static type checking enables the compiler to verify the correctness of your program without running it. The following code will fail because the compiler enforces static typing:

```
int x = "5";
```

```
int x = "5";
```

Runtime type checking is performed by the CLR when you downcast via a reference conversion or unboxing. For example:

```
object y = "5";
int z = (int) y;

// Runtime error, downcast failed
```

Runtime type checking is possible because each object on the heap internally stores a little type token. This token can be retrieved by calling the `GetType` method of `object`.

The GetType Method and typeof Operator

Creating Types

All types in C# are represented at runtime with an instance of `System.Type`. There are two basic ways to get a `System.Type` object:

- Call `GetType` on the instance.
- Use the `typeof` operator on a type name.