C#

The `foreach` statement iterates over an *enumerable* object. An enumerable object is the logical representation of a sequence. It is not itself a cursor, but an object that produces cursors over itself. An enumerable object either:

- Implements `IEnumerable` or `IEnumerable<T>`
- Has a method named `GetEnumerator` that returns an *enumerator*

`IEnumerator` and `IEnumerable` are defined in `System.Collections`. `IEnumerator<T>` and `IEnumerable<T>` are defined in `System.Collections.Generic`.

The enumeration pattern is as follows:

The enumeration pattern is as follows:

```
class Enumerator   // Typically implements IEnumerator or IEnumerator<T>
{
  public IteratorVariableType Current { get {...} }
  public bool MoveNext() {...}
}
```



```
class Enumerable   // Typically implements IEnumerable or IEnumerable<T>
{
  public Enumerator GetEnumerator() {...}
}
```

Here is the high-level way of iterating through the characters in the word *beer* using a foreach statement:

```
foreach (char c in "beer")
  Console.WriteLine (c);
```

Here is the low-level way of iterating through the characters in *beer* without using a `foreach` statement:

```
using (var enumerator = "beer".GetEnumerator())
  while (enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine (element);
  }
```

If the enumerator implements `IDisposable`, the `foreach` statement also acts as a `using` statement, implicitly disposing the enumerator object as in the earlier example.

Chapter 7 explains the enumeration interfaces in further detail.

Collection Initializers

You can instantiate and populate an enumerable object in a single step. For example:

```
using System.Collections.Generic;
...

    List<int> list = new List<int> {1, 2, 3};
```

The compiler translates this to the following:

```
using System.Collections.Generic;
...
List<int> list = new List<int>();
list.Add (1);
list.Add (2);
list.Add (3);
```

```
    list.Add (2);
    list.Add (3);
```

This requires that the enumerable object implements the `System.Collections.IEnumerable` interface, and that it has an `Add` method that has the appropriate number of parameters for the call.

Iterators

Whereas a `foreach` statement is a *consumer* of an enumerator, an iterator is a *producer* of an enumerator. In this example, we use an iterator to return a sequence of Fibonacci numbers (where each number is the sum of the previous two):

```
using System;
using System.Collections.Generic;

class Test
```

```
class Test
{
  static void Main()
  {
    foreach (int fib in Fibs(6))
      Console.Write (fib + "  ");
  }
```

```
static IEnumerable<int> Fibs (int fibCount)
{
  for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
  {
    yield return prevFib;
    int newFib = prevFib+curFib;
    prevFib = curFib;
```

```
        int newFib = prevFib+curFib;
        prevFib = curFib;
        curFib = newFib;
      }
    }
  }
```


OUTPUT: 1

```
OUTPUT: 1
1
2
3
5
8
```

Whereas a `return` statement expresses "Here's the value you asked me to return from this method," a `yield return` statement expresses "Here's the next element you asked me to yield from this enumerator." On each `yield` statement, control is returned to the caller, but the callee's state is maintained so that the method can continue executing as soon as the caller enumerates the next element. The lifetime of this state is bound to the enumerator, such that the state can be released when the caller has finished enumerating.

the caller has finished enumerating.

> The compiler converts iterator methods into private classes that implement `IEnumerable<T>` and/or `IEnumerator<T>`. The logic within the iterator block is "inverted" and spliced into the `MoveNext` method and `Current` property on the compiler-written enumerator class. This means that when you call an iterator method, all you're doing is instantiating the compiler-written class; none of your code actually runs! Your code runs only when you start enumerating over the resultant sequence, typically with a `foreach` statement.

Iterator Semantics

An iterator is a method, property, or indexer that contains one or more `yield` statements. An iterator must return one of the following four interfaces (otherwise, the compiler will generate an error):

compiler will generate an error):

```
// Enumerable interfaces
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>
```

```
// Enumerator interfaces
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

An iterator has different semantics, depending on whether it returns an *enumerable* interface or an *enumerator* interface. We describe this in Chapter 7.

Multiple yield statements are permitted. For example:

```
class Test
{
  static void Main()
  {
```

```
  {
    foreach (string s in Foo())
      Console.WriteLine(s);
  }

// Prints "One","Two","Three"

  static IEnumerable<string> Foo()
  {
    yield return "One";
    yield return "Two";
    yield return "Three";
  }
```

```
    yield return "Three";
  }
}
```

yield break

The `yield break` statement indicates that the iterator block should exit early, without returning more elements. We can modify `Foo` as follows to demonstrate:

```
static IEnumerable<string> Foo (bool breakEarly)
{
  yield return "One";
  yield return "Two";

  if (breakEarly)
    yield break;

  yield return "Three";
```

```
    yield return "Three";
}
```

A `return` statement is illegal in an iterator block—you must use a `yield break` instead.

Iterators and try/catch/finally blocks

A `yield return` statement cannot appear in a `try` block that has a `catch` clause:

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }    // Illegal
  catch { ... }
}
```

```
}
```

Nor can `yield return` appear in a `catch` or `finally` block. These restrictions are due to the fact that the compiler must translate iterators into ordinary classes with `MoveNext`, `Current`, and `Dispose` members, and translating exception handling blocks would create excessive complexity.

You can, however, yield within a `try` block that has (only) a `finally` block:

```
IEnumerable<string> Foo()
{
  try { yield return "One"; }    // OK
  finally { ... }
}
```

The code in the `finally` block executes when the consuming enumerator reaches the end of the sequence or is disposed. A `foreach` statement implicitly disposes the enumerator if you break early, making this a safe way to consume enumerators. When working with enumerators explicitly, a trap is to abandon enumeration early without disposing it, circumventing the `finally` block. You can avoid this risk by wrapping explicit use of enumerators in a `using` statement:

[redacted]

```
string firstElement = null;
var sequence = Foo();
using (var enumerator = sequence.GetEnumerator())
  if (enumerator.MoveNext())
    firstElement = enumerator.Current;
```

Composing Sequences

Composing Sequences

Iterators are highly composable. We can extend our example, this time to output even Fibonacci numbers only:

```
using System;
using System.Collections.Generic;
```

```
class Test
{
  static void Main()
  {
    foreach (int fib in EvenNumbersOnly (Fibs(6)))
      Console.WriteLine (fib);
  }
```

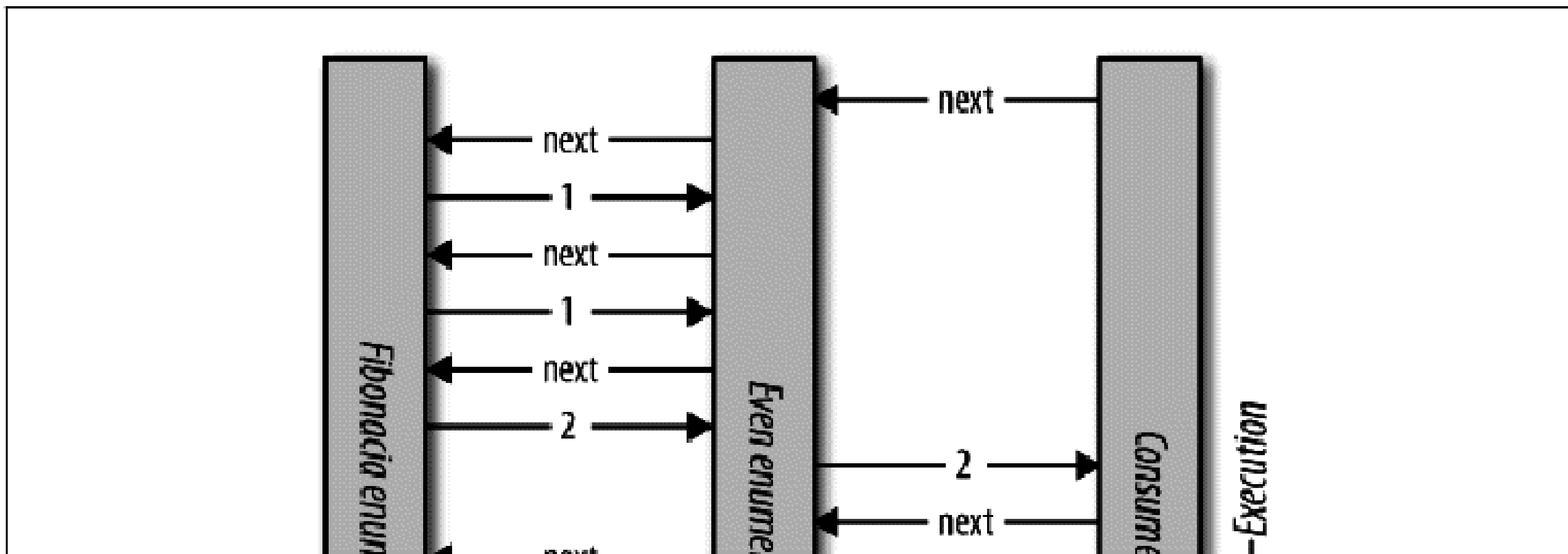
```
}
static IEnumerable<int> Fibs (int fibCount)
{
  for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
  {
```



```
    yield return prevFib;
    int newFib = prevFib+curFib;
    prevFib = curFib;
    curFib = newFib;
  }
}
```

```
static IEnumerable<int> EvenNumbersOnly (IEnumerable<int> sequence)
  {
    foreach (int x in sequence)
      if ((x % 2) == 0)
        yield return x;
  }
}
```

Each element is not calculated until the last moment—when requested by a `MoveNext()` operation. Figure 4-1 shows the data requests and data output over time.



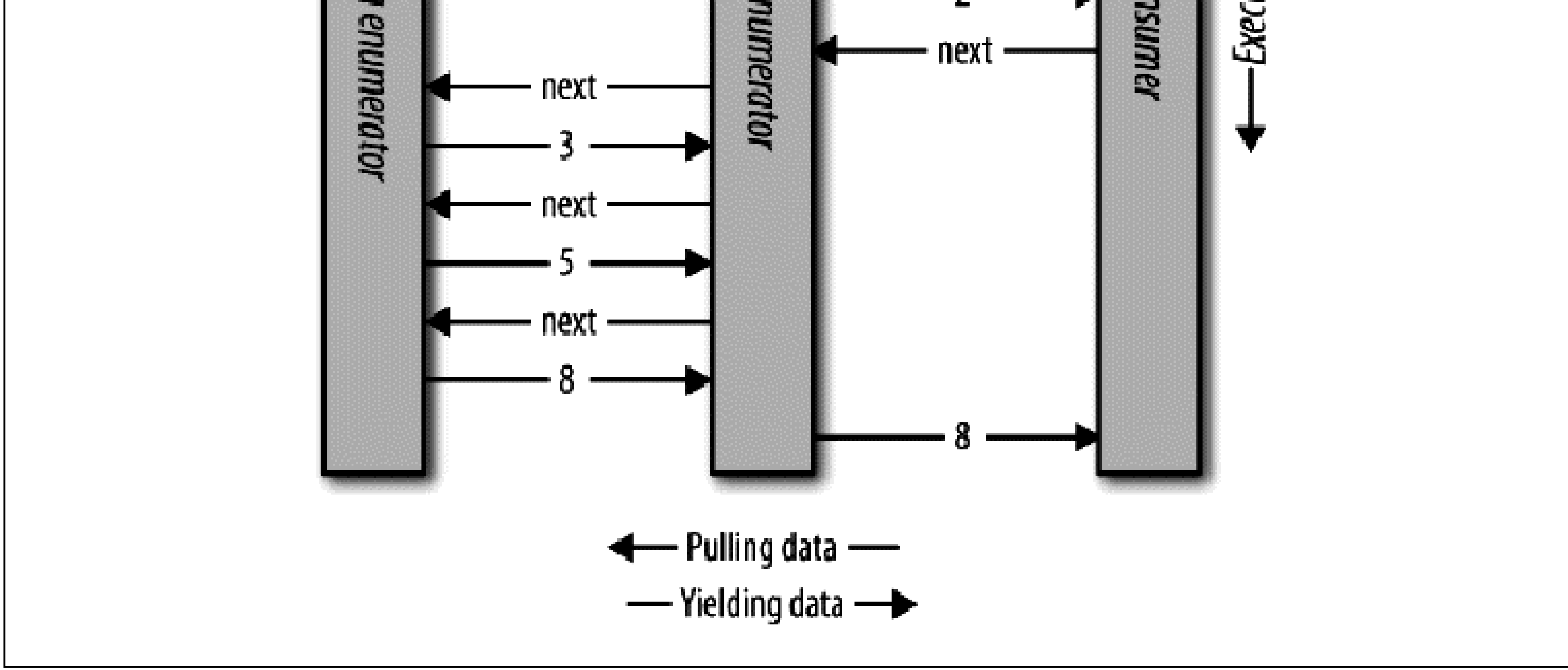


Figure 4-1. Composing sequences

The composability of the iterator pattern is extremely useful in LINQ; we will discuss the subject again in Chapter 8.

Nullable Types

Reference types can represent a nonexistent value with a null reference. Value types, however, cannot ordinarily represent null values. For example:

```
string s = null;        // OK, Reference Type
int i = null;           // Compile Error, Value Type cannot be null
```


To represent null in a value type, you must use a special construct called a *nullable type*. A nullable type is denoted with a value type followed by the ? symbol:

```
int? i = null;                      // OK, Nullable Type
Console.WriteLine (i == null);      // True
```

Nullable<T> Struct

T? translates into System.Nullable<T>. Nullable<T> is a lightweight immutable structure, having only two fields, to represent Value and HasValue. The essence of System.Nullable<T> is very simple:

```
public struct Nullable<T> where T : struct
{
```

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault();
  public T GetValueOrDefault (T defaultValue);
  ...
}
```

The code:

```
int? i = null;
Console.WriteLine (i == null);
// True
```

```
// True
// True
```

translates to:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);
```

#

Attempting to retrieve `Value` when `HasValue` is false throws an `InvalidOperationException`. `GetValueOrDefault()` returns `Value` if `HasValue` is true; otherwise, it returns `new T()` or a specified custom default value.

The default value of `T?` is `null`.

Implicit and Explicit Nullable Conversions

The conversion from `T` to `T?` is implicit, and from `T?` to `T` is explicit. For example:

```
int? x = 5;         // implicit
int y = (int)x;     // explicit
```

The explicit cast is directly equivalent to calling the nullable object's `Value` property. Hence, an `InvalidOperationException` is thrown if `HasValue` is false.

Boxing and Unboxing Nullable Values

Boxing and Unboxing Nullable Values

When T? is boxed, the boxed value on the heap contains T, not T?. This optimization is possible because a boxed value is a reference type that can already express null.

C# also permits the unboxing of nullable types with the as operator. The result will be null if the cast fails:


```
object o = "string";
int? x = o as int?;
Console.WriteLine (x.HasValue);
```

```
// False
```

Operator Lifting

Operator Lifting

The `Nullable<T>` struct does not define operators such as `<`, `>`, or even `==`. Despite this, the following code compiles and executes correctly:

```
int? x = 5;
int? y = 10;
bool b = x < y;
```

```
// true
```

This works because the compiler steals or "lifts" the less-than operator from the underlying value type. Semantically, it translates the preceding comparison expression into this:

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

In other words, if both `x` and `y` have values, it compares via `int`'s less-than operator; otherwise, it returns `false`.

Operator lifting means you can implicitly use `T`'s operators on `T?`. You can define operators for `T?` in order to provide special-purpose null behavior, but in the vast majority of cases, it's best to rely on the compiler automatically applying systematic nullable logic for you. Here are some examples:

```
int? x = 5;
int? y = null;
```

```
// Equality operator examples
Console.WriteLine (x == y);    // False
Console.WriteLine (x == null); // False
Console.WriteLine (x == 5);    // True
Console.WriteLine (y == null); // True
```

```
Console.WriteLine (y == null); // True
Console.WriteLine (y == 5);    // False
Console.WriteLine (y != 5);    // True

// Relational operator examples
Console.WriteLine (x < 6);     // True
Console.WriteLine (y < 6);     // False
Console.WriteLine (y > 6);     // False
```

```
// All other operator examples
Console.WriteLine (x + 5);     // 10
Console.WriteLine (x + y);     // null (prints empty line)
```

The compiler performs null logic differently depending on the category of operator. The following sections explain these different rules.

Equality operators (== !=)

Lifted equality operators handle nulls just like reference types do. This means two null values are equal:

```
Console.WriteLine (       null ==        null);   // True
Console.WriteLine ((bool?)null == (bool?)null);   // True
```


Further:

-
-

If exactly one operand is null, the operands are unequal.

If both operands are non-null, their `Values` are compared.

Relational operators (< <= >= >)

Relational operators (< <= >= >)

The relational operators work on the principle that it is meaningless to compare null operands. This means comparing a null value to either a null or a non-null value returns false.

```
bool b = x < y;    // Translation:
bool b = (x == null || y == null) ? false : (x.Value < y.Value);
```

```
// b is false (assuming x is 5 and y is null)
```

All other operators (+ − * / % & | ^ << >> + ++ -- ! ~)

These operators return null when any of the operands are null. This pattern should be familiar to SQL users.

```
int? c = x + y;   // Translation:
```

```
int? c = (x == null || y == null)
         ? null
         : (int?) (x.Value + y.Value);
```

```
            : null
            : (int?) (x.Value + y.Value);
// c is null (assuming x is 5 and y is null)
```

Advanced C#

An exception is when the & and | operators are applied to bool?, which we will discuss shortly.

Mixing nullable and non-nullable operators

You can mix and match nullable and non-nullable types (this works because there is an implicit conversion from `T` to `T?`):

```
int? a = null;
int b = 2;
int? c = a + b;

// c is null - equivalent to a + (int?)b
```

bool? with & and | Operators

When supplied operands of type `bool?`, the `&` and `|` operators treat `null` as an *unknown value*. So, `null | true` is true, because:

•

If the unknown value is false, the result would be true.

If the unknown value is true, the result would be true.

Similarly, `null & false` is false. This behavior would be familiar to SQL users. The following example enumerates other combinations:

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine (n | n);
// (null)
```

```
// (null)
```



```
Console.WriteLine (n | f);
Console.WriteLine (n | t);
Console.WriteLine (n & n);
Console.WriteLine (n & f);
Console.WriteLine (n & t);
```

```
// (null)
// True
// (null)
// False
```

```
// False
// (null)
```

Null Coalescing Operator

The ?? operator is the null coalescing operator, and it can be used with both nullable types and reference types. It says, "If the operand is non-null, give it to me; otherwise, give me a default value." For example:

```
int? x = null;
int y = x ?? 5;
```

```
// y is 5
```

```
int? a = null, b = 1, c = 2;
```

```
int? a = null, b = 1, c = 2;
Console.WriteLine (a ?? b ?? c);
```

```
// 1 (first non-null value)
```

The ?? operator is equivalent to calling `GetValueOrDefault` with an explicit default value, except that the expression passed to `GetValueOrDefault` is never evaluated if the variable is not null.

Scenarios for Nullable Types

One of the most common scenarios for nullable types is to represent unknown values. This frequently occurs in database programming, where a class is mapped to a table with nullable columns. If these columns are strings (e.g., an EmailAddress column on a Customer table), there is no problem, as string is a reference type in the CLR, which can be null. However, most other SQL column types map to CLR struct types, making nullable types very useful when mapping SQL to the CLR. For example:

example:

```
// Maps to a Customer table in a database
public class Customer
{
  ...
  public decimal? AccountBalance;
}
```

A nullable type can also be used to represent the backing field of what's sometimes called an *ambient property*. An ambient property, if null, returns the value of its parent. For example:

```
public class Row
{
...
Grid parent;
Color? color;
```

```
  public Color Color
  {
    get { return color ?? parent.Color; }
    set { color = Color == parent.Color ? (Color?)null : value; }
  }
}
```


Alternatives to Nullable Types

Before nullable types were part of the C# language (i.e., before C# 2.0), there were many strategies to deal with nullable value types, examples of which still appear in the .NET Framework for historical reasons. One of these strategies is to designate a particular non-null value as the "null value"; an example is in the string and array classes. `String.IndexOf` returns the magic value of `-1` when the character is not found:

```
int i = "Pink".IndexOf ('b');
Console.WriteLine (s);          // -1
```

```
Console.WriteLine (s);          // -1
```

However, `Array.IndexOf` returns `-1` only if the index is 0-bounded. The more general formula is that `IndexOf` returns 1 less than the lower bound of the array. In the next example, `IndexOf` returns `0` when an element is not found:

```
// Create an array whose lower bound is 1 instead of 0:
```

```
Array a = Array.CreateInstance (typeof (string),
                                new int[] {2}, new int[] {1});
a.SetValue ("a", 1);
a.SetValue ("b", 2);
Console.WriteLine (Array.IndexOf (a, "c"));  // 0
```

Nominating a “magic value” is problematic for several reasons:

It means that each value type has a different representation of null. In contrast, nullable types provide one common pattern that works for all value types.

There may be no reasonable designated value. In the previous example, –1 could not always be used. The same is true for our earlier examples representing an unknown account balance and an unknown temperature.

Forgetting to test for the magic value results in an incorrect value that may go unnoticed until later in execution—when it pulls an unintended magic trick. Forgetting to test `HasValue` on a null value, however, throws an `InvalidOperationException` on the spot.

The ability for a value to be null is not captured in the *type*. Types communicate the intention of a program, allow the compiler to check for correctness, and enable a consistent set of rules enforced by the compiler.

Operator Overloading

Operators can be overloaded to provide more natural syntax for custom types. Operator overloading is most appropriately used for implementing custom structs that represent fairly primitive data types. For example, a custom numeric type is an excellent candidate for operator overloading.

The following symbolic operators can be overloaded:

+ (unary)

+ (unary)

--

%

- (unary)

+

&

!

-

|

~

*

^

++

/

++

/

<<

`>>` `==` `!=` `>` `<`

`>=` `<=`

The following operators are also overloadable:

Implicit and explicit conversions (with the `implicit` and `explicit` keywords)

The literals `true` and `false`

The following operators are indirectly overloaded:

The compound assignment operators (e.g., +=, /=) are implicitly overridden by overriding the noncompound operators (e.g., +, /).

The conditional operators && and || are implicitly overridden by overriding the bitwise operators & and |.

Operator Functions

An operator is overloaded by declaring an *operator function*. An operator function has the following rules:

The name of the function is specified with the **operator** keyword followed by an operator symbol.

The operator function must be marked **static**.

The parameters of the operator function represent the operands.

The return type of an operator function represents the result of an expression.

At least one of the operands must be the type in which the operator function is declared.

In the following example, we define a struct called Note representing a musical note, and then overload the + operator:

```
public struct Note
{
  int value;
  public Note (int semitonesFromA) { value = semitonesFromA; }
  public static Note operator + (Note x, int semitones)
  {
    return new Note (x.value + semitones);
```

```
    return new Note (x.value + semitones);
  }
}
```

This overload allows us to add an `int` to a `Note`:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Overloading an assignment operator automatically supports the corresponding compound assignment operator. In our example, since we overrode `+`, we can use `+=` too:

```
CSharp += 2;
```


Overloading Equality and Comparison Operators

Equality and comparison operators are sometimes overridden when writing structs, and in rare cases when writing classes. Special rules and obligations come with overloading the equality and comparison operators, which we explain in Chapter 6. A summary of these rules is as follows:

Pairing
: The C# compiler enforces operators that are logical pairs to both be defined. These operators are (== !=), (< >), and (<= >=).

`Equals` *and* `GetHashCode`
: In most cases, if you overload (==) and (!=), you will usually need to override the `Equals` and `GetHashCode` methods defined on `object` in order to get meaningful behavior. The C# compiler will give a warning if you do not do this. (See "Equality Comparison" on page 245 in Chapter 6 for more details.)

`IComparable` *and* `IComparable<T>`
: If you overload (< >) and (<= >=), you should implement `IComparable` and `IComparable<T>`.

Custom Implicit and Explicit Conversions

Implicit and explicit conversions are overloadable operators. These conversions are typically overloaded to make converting between strongly related types (such as numeric types) concise and natural.

To convert between weakly related types, the following strategies are more suitable:

- Write a constructor that has a parameter of the type to convert from.
- Write `To`*XXX* and (static) `From`*XXX* methods to convert between types.

As explained in the discussion on types, the rationale behind implicit conversions is that they are guaranteed to succeed and do not lose information during the conversion. Conversely, an explicit conversion should be required either when runtime circumstances will determine whether the conversion will succeed or if information may be lost during the conversion.

In this example, we define conversions between our musical `Note` type and a double (which represents the frequency in hertz of that note):

```
...
// Convert to hertz
public static implicit operator double (Note x)
{
  return 440 * Math.Pow (2, (double) x.value / 12 );
}
```

```
  }

// Convert from hertz (accurate to the nearest semitone)
public static explicit operator Note (double x)
{
  return new Note ((int) (0.5 + 12 * (Math.Log (x/440) / Math.Log(2) ) ));
}
...
```



```
Note n = (Note)554.37;
double x = n;
```

```
// explicit conversion
// implicit conversion
```

Following our own guidelines, this example might be

Following our own guidelines, this example might be better implemented with a `ToFrequency` method (and a static `FromFrequency` method) instead of implicit and explicit operators.

Custom conversions are ignored by the `as` and `is` operators:

```
Console.WriteLine (554.37 is Note);
Note n = 554.37 as Note;
```

```
// False
```

```
// False
// Error
```

Overloading true and false

The `true` and `false` operators are overloaded in the extremely rare case of types that are boolean "in spirit," but do not have a conversion to `bool`. An example is a type that implements three-state logic: by overloading `true` and `false`, such a type can work seamlessly with conditional statements and operators—namely, `if`, `do`, `while`, `for`, `&&`, `||`, and `?:`. The `System.Data.SqlTypes.SqlBoolean` struct provides this functionality. For example:

```
SqlBoolean a = SqlBoolean.Null;
if (a)
  Console.WriteLine ("True");
else if (!a)
```

```
else if (!a)
  Console.WriteLine ("False");
else
  Console.WriteLine ("Null");
```

```
OUTPUT:
Null
```

The following code is a reimplementation of the parts of `SqlBoolean` necessary to demonstrate the `true` and `false` operators:

```
public struct SqlBoolean
{
  public static bool operator true (SqlBoolean x)
  {
```

```
  {
    return x.m_value == True.m_value;
  }

public static bool operator false (SqlBoolean x)
{
  return x.m_value == False.m_value;
}
 public static SqlBoolean operator ! (SqlBoolean x)
{
  if (x.m_value == Null.m_value)  return Null;
```



```
  if (x.m_value == False.m_value) return True;
  return False;
}
```

```
}

public static readonly SqlBoolean Null =  new SqlBoolean(0);
public static readonly SqlBoolean False = new SqlBoolean(1);
public static readonly SqlBoolean True =  new SqlBoolean(2);

  private SqlBoolean (byte value) { m_value = value; }
  private byte m_value;
}
```

Extension Methods

Extension methods allow an existing type to be extended with new methods without altering the definition of the original type. An extension method is a static method of a static class, where the `this` modifier is applied to the first parameter. The type of the first parameter will be the type that is extended. For example:

```
public static class StringHelper
{
```

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

#

The IsCapitalized extension method can be called as though it were an instance method on a string, as follows:

```
Console.WriteLine ("Perth".IsCapitalized());
```

An extension method call, when compiled, is translated back into an ordinary static method call:

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

The translation works as follows:

```
arg0.Method (arg1, arg2, ...);               // Extension method call
StaticClass.Method (arg0, arg1, arg2, ...); // Static method call
```

Interfaces can be extended too:

Interfaces can be extended too.

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;

  throw new InvalidOperationException ("No elements!");
}
...
Console.WriteLine ("Seattle".First());   // S
```

Extension methods were added in C# 3.0.

Extension Method Chaining

Extension methods, like instance methods, provide a tidy way to chain functions.

Extension methods, like instance methods, provide a tidy way to chain functions. Consider the following two functions:

```
public static class StringHelper
{
  public static string Pluralize (this string s) {...}
  public static string Capitalize (this string s) {...}
}
```

`x` and `y` are equivalent and both evaluate to `"Sausages"`, but `x` uses extension methods, whereas `y` uses static methods:

```
string x = "sausage".Pluralize().Capitalize();
string y = StringHelper.Capitalize (StringHelper.Pluralize ("sausage"));
```

Ambiguity and Resolution

Namespaces

An extension method cannot be accessed unless the namespace is in scope. Consider the extension method `IsCapitalized` in the following example:

An extension method cannot be accessed unless the namespace is in scope. Consider the extension method `IsCapitalized` in the following example:

```
using System;
```

```
namespace Utils
{
  public static class StringHelper
  {
    public static bool IsCapitalized (this string s)
    {
      if (string.IsNullOrEmpty(s)) return false;
      return char.IsUpper (s[0]);
    }
  }
}
```

To use `IsCapitalized`, the following application must import `Utils`, in order to avoid a compile-time error:

To use `IsCapitalized`, the following application must import `Utils`, in order to avoid a compile-time error:

```
namespace MyApp
{
  using Utils;

  class Test
  {
    static void Main()
    {
      Console.WriteLine ("Perth".IsCapitalized());
    }
  }
}
```

Extension methods versus instance methods

Any compatible instance method will always take precedence over an extension method. In the following example, `Test`'s `Foo` method will always take precedence—even when called with an argument `x` of type `int`:

```
class Test
{
  public void Foo (object x) { }
}
// This method always wins

static class Extensions
```

```
static class Extensions
{
  public static void Foo (this Test t, int x) { }
}
```

The only way to call the extension method in this case is via normal static syntax; in other words, `Extensions.Foo(...)`.

Extension methods versus extension methods

If two extension methods have the same signature, the extension method must be called as an ordinary static method to disambiguate the method to call. If one extension method has more specific arguments, however, the more specific method takes precedence.

To illustrate, consider the following two classes:

```
static class StringHelper
{
  public static bool IsCapitalized (this string s) {...}
}
static class ObjectHelper
{
  public static bool IsCapitalized (this object s) {...}
}
```

```
}
```

The following code calls `StringHelper`'s `IsCapitalized` method:

```
bool test1 = "Perth".IsCapitalized();
```

To call `ObjectHelper`'s `IsCapitalized` method, we must specify it explicitly:

```
bool test2 = (ObjectHelper.IsCapitalized ("Perth"));
```

Concrete types are considered more specific than interfaces.

Extension Methods on Interfaces

Extension methods can apply to interfaces:

```
using System;
using System.Collections.Generic;
static class Test
{
```

```
static class Test
{
```



```
static void Main()
{
  string[] strings = { "a", "b", null, "c"};
  foreach (string s in strings.StripNulls())
    Console.WriteLine (s);
}
```

```
static IEnumerable<T> StripNulls<T> (this IEnumerable<T> seq)
{
  foreach (T t in seq)
    if (t != null)
      yield return t;
}
```

```
  }
}
```

Anonymous Types

An anonymous type is a simple class created by the compiler on the fly to store a set of values. To create an anonymous type, use the new keyword followed by an object initializer, specifying the properties and values the type will contain. For example:

```
var dude = new { Name = "Bob", Age = 1 };
```

The compiler translates this to (approximately) the following:

```
internal class AnonymousGeneratedTypeName
{
  private string name;  // Actual field name is irrelevant
  private int    age;   // Actual field name is irrelevant

public AnonymousGeneratedTypeName (string name, int age)
```

```
public AnonymousGeneratedTypeName (string name, int age)
{
  this.name = name; this.age = age;
}

public string
public int

Name { get { return name; } }
Age  { get { return age;  } }

  // The Equals and GetHashCode methods are overridden (see Chapter 6).
  // The ToString method is also overridden.
}
...

var dude = new AnonymousGeneratedTypeName ("Bob", 1);
```

```
var dude = new AnonymousGeneratedTypeName ("Bob", 1);
```

You must use the var keyword to reference an anonymous type, because the name of the type is compiler-generated.

The property name of an anonymous type can be inferred from an expression that is itself an identifier (or ends with one). For example:

```
int Age = 23;
var dude = new { Name = "Bob", Age, Age.ToString().Length };
```

is equivalent to:

```
var dude = new { Name = "Bob", Age = Age, Length = Age.ToString().Length };
```


Two anonymous type instances will have the same underlying type if their elements are same-typed and they're declared within the same assembly:

```
var a1 = new { X = 2, Y = 4 };
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType());   // True
```

```
var a2 = new { X = 2, Y = 4 };
Console.WriteLine (a1.GetType() == a2.GetType());   // True
```

Additionally, the `Equals` method is overridden to perform equality comparisons:

```
Console.WriteLine (a1 == a2);          // False
Console.WriteLine (a1.Equals (a2));    // True
```

Anonymous types are used primarily when writing LINQ queries (see Chapter 8), and were added in C# 3.0.

Dynamic Binding

Dynamic binding defers *binding*—the process of resolving types, members, and operations—from compile time to runtime. Dynamic binding is useful when at compile time *you* know that a certain function, member, or operation exists, but the *compiler* does not. This commonly occurs when you are interoperating with dynamic languages (such as IronPython) and COM and in scenarios when you might otherwise use reflection.

A dynamic type is declared with the contextual keyword `dynamic`:

A dynamic type is declared with the contextual keyword `dynamic`:

```
dynamic d = GetSomeObject();
d.Quack();
```


A dynamic type tells the compiler to relax. We expect the runtime type of `d` to have a `Quack` method. We just can't prove it statically. Since `d` is dynamic, the compiler defers binding `Quack` to `d` until runtime. To understand what this means requires distinguishing between *static binding* and *dynamic binding*.

defers binding `Quack` to `d` until runtime. To understand what this means requires distinguishing between *static binding* and *dynamic binding*.

Static Binding Versus Dynamic Binding

The canonical binding example is mapping a name to a specific function when compiling an expression. To compile the following expression, the compiler needs to find the implementation of the method named `Quack`:

```
d.Quack();
```

Let's suppose the static type of `d` is `Duck`:

```
Duck d = ...
d.Quack();
```

In the simplest case, the compiler does the binding by looking for a parameterless method named `Quack` on `Duck`. Failing that, the compiler extends its search to methods taking optional parameters, methods on base classes of `Duck`, and extension methods that take `Duck` as its first parameter. If no match is found, you'll get a com-methods that take `Duck` as its first parameter. If no match is found, you'll get a compilation error. Regardless of what method gets bound, the bottom line is that the binding is done by the compiler, and the binding utterly depends on statically knowing the types of the operands (in this case, `d`). This makes it *static binding*.

Now let's change the static type of `d` to `object`:

```
object d = ...
d.Quack();
```

Calling `Quack` gives us a compilation error, because although the value stored in `d` can contain a method called `Quack`, the compiler cannot know it since the only information it has is the type of the variable, which in this case is `object`. But let's now change the static type of `d` to `dynamic`:

```
dynamic d = ...
d.Quack();
```

```
d.Quack();
```

A `dynamic` type is like `object`—it's equally nondescriptive about a type. The difference is that it lets you use it in ways that aren't known at compile time. A dynamic object binds at runtime based on its runtime type, not its compile-time type. When the compiler sees a dynamically bound expression (which in general is an expression that contains any value of type `dynamic`), it merely packages up the expression such that the binding can be done later at runtime.

At runtime, if a dynamic object implements `IDynamicMetaObjectProvider`, that interface is used to perform the binding. If not, binding occurs in almost the same way as it would have had the compiler known the dynamic object's runtime type. These two alternatives are called *custom binding* and *language binding*.

> COM interop can be considered to use a third kind of binding (see Chapter 25).

Custom Binding

Custom Binding

Custom binding occurs when a dynamic object implements `IDynamicMetaObjectProvider` (IDMOP). Although you can implement IDMOP on types that you write in C#, and that is useful to do, the more common case is that you have acquired an IDMOP object from a dynamic language that is implemented in .NET on the DLR, such as IronPython or IronRuby. Objects from those languages implicitly implement IDMOP as a means by which to directly control the meanings of operations performed on them.

We will discuss custom binders in greater detail in Chapter 19, but we will write a simple one now to demonstrate the feature:

```
using System;
using System.Dynamic;

public class Test
{
```

```
{
  static void Main()
  {
    dynamic d = new Duck();
    d.Quack();
    d.Waddle();
```

```
// Quack method was called
// Waddle method was called
```



```
  }
}
```

```
    }
  }
```

```
public class Duck : DynamicObject
{
  public override bool TryInvokeMember (
    InvokeMemberBinder binder, object[] args, out object result)
  {
    Console.WriteLine (binder.Name + " method was called");
    result = null;
    return true;
  }
}
```

The `Duck` class doesn't actually have a `Quack` method. Instead, it uses custom binding to intercept and interpret all method calls.

Language Binding

Language binding occurs when a dynamic object does not implement `IDynamicMetaObjectProvider`. Language binding is useful when working around imperfectly designed types or inherent limitations in the .NET type system (we'll explore more scenarios in Chapter 19). A typical problem when using numeric types is that they have no common interface. We have seen that methods can be bound dynamically; the same is true for operators:

```
static dynamic Mean (dynamic x, dynamic y)
{
  return (x + y) / 2;
}
```

C#

```
static void Main()
{
  int x = 3, y = 4;
  Console.WriteLine (Mean (x, y));
}
```

The benefit is obvious—you don't have to duplicate code for each numeric type. However, you lose static type safety, risking runtime exceptions rather than compile-time errors.

Dynamic binding circumvents static type safety, but not run-time type safety. Unlike with reflection (Chapter 18), you can't

Dynamic binding circumvents static type safety, but not runtime type safety. Unlike with reflection (Chapter 18), you can't circumvent member accessibility rules with dynamic binding.

By design, language runtime binding behaves as similarly as possible to static binding, had the runtime types of the dynamic objects been known at compile time. In our previous example, the behavior of our program would be identical if we hardcoded `Mean` to work with the `int` type. The most notable exception in parity between static and dynamic binding is for extension methods, which we discuss in "Uncallable Functions" on page 168.

Dynamic binding also incurs a performance hit. Because of the DLR's caching mechanisms, however, repeated calls to the same dynamic expression are optimized—allowing you to efficiently call dynamic expressions in a loop. This optimization brings the typical overhead for a simple dynamic expression on today's hardware down to less than 100 ns.

RuntimeBinderException

If a member fails to bind, a `RuntimeBinderException` is thrown. You can think of this like a compile-time error at runtime:

```
dynamic d = 5;
d.Hello();
```

```
// throws RuntimeBinderException
```

The exception is thrown because the `int` type has no `Hello` method.

Runtime Representation of Dynamic

There is a deep equivalence between the `dynamic` and `object` types. The runtime

There is a deep equivalence between the dynamic and object types. The runtime treats the following expression as true:

```
typeof (dynamic) == typeof (object)
```

This principle extends to constructed types and array types:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Like an object reference, a dynamic reference can point to an object of any type (except pointer types):

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name);

// String

x = 123;  // No error (despite same variable)
```

```
x = 123;  // No error (despite same variable)
Console.WriteLine (x.GetType().Name);  // Int32
```

Structurally, there is no difference between an object reference and a dynamic reference. A dynamic reference simply enables dynamic operations on the object it points to. You can convert from `object` to `dynamic` to perform any dynamic operation you want on an `object`:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("hello");
Console.WriteLine (o);   // hello
```


Reflecting on a type exposing (public) `dynamic` members reveals that those members are represented as annotated `objects`. For example:

```
public class Test
{
  public dynamic Foo;
```

```
  public dynamic Foo;
}
```

is equivalent to:

```
public class Test
{
  [System.Runtime.CompilerServices.DynamicAttribute]
  public object Foo;
}
```

This allows consumers of that type to know that `Foo` should be treated as dynamic, while allowing languages that don't support dynamic binding to fall back to `object`.

Dynamic Conversions

The dynamic type has implicit conversions to and from all other types:*

```
int i = 7;
dynamic d = i;
int j = d;

// No cast required (implicit conversion)
```

For the conversion to succeed, the runtime type of the dynamic object must be implicitly convertible to the target static type. The preceding example worked because an `int` is implicitly convertible to a `long`.

The following example throws a `RuntimeBinderException` because an `int` is not implicitly convertible to a `short`:

```
int i = 7;
dynamic d = i;
short j = d;
```

```
// throws RuntimeBinderException
```

```
// throws RuntimeBinderException
```

var Versus dynamic

The var and dynamic types bear a superficial resemblance, but the difference is deep:

-
-

var says, "Let the *compiler* figure out the type."

dynamic says, "Let the *runtime* figure out the type."

```
dynamic x = "hello";
var y = "hello";
int i = x;
```

```
int i = x;
int j = y;
```

```
// Static type is dynamic, runtime type is string
// Static type is string, runtime type is string
// Runtime error
// Compile-time error
```

* Technically, the conversion from dynamic to other types is not an *implicit conversion*, but an *assignment conversion*. An assignment conversion is more restrictive in situations that might otherwise create ambiguities, such as overload resolution.

The static type of a variable declared with var can be dynamic:

```
dynamic x = "hello";
var y = x;
```

```
var y = x;
int z = y;
```

```
// Static type of y is dynamic
// Runtime error
```

Dynamic Expressions

Fields, properties, methods, events, constructors, indexers, operators, and conversions can all be called dynamically.

Trying to consume the result of a dynamic expression with a `void` return type is prohibited—just as with a statically typed expression. The difference is that the error occurs at runtime:

```
dynamic list = new List<int>();
```

```
dynamic list = new List<int>();
var result = list.Add (5);
```

```
// RuntimeBinderException thrown
```

Expressions involving dynamic operands are typically themselves dynamic, since the effect of absent type information is cascading:

```
dynamic x = 2;
var y = x * 3;
```

```
// Static type of y is dynamic
```

There are a couple of obvious exceptions to this rule. First, casting a dynamic expression to a static type yields a static expression:

```
dynamic x = 2;
```

```
dynamic x = 2;
var y = (int)2;
```

```
// Static type of y is int
```

Second, constructor invocations always yield static expressions—even when called with dynamic arguments. In this example, `x` is statically typed to a `StringBuilder`:

```
dynamic capacity = 10;
var x = new System.Text.StringBuilder (capacity);
```

In addition, there are a few edge cases where an expression containing a dynamic argument is static, including passing an index to an array and delegate creation expressions.

Dynamic Calls Without Dynamic Receivers

The canonical use case for `dynamic` involves a dynamic *receiver*. This means that a dynamic object is the receiver of a dynamic function call:

```
dynamic x = ...;
x.Foo();
```

```
// x is the receiver
```

However, you can also call statically known functions with dynamic arguments. Such calls are subject to dynamic overload resolution, and can include:

-
-
-

Static methods

Instance constructors

Instance methods on receivers with a statically known type

In the following example, the particular Foo that gets dynamically bound is dependent on the runtime type of the dynamic argument:

```
class Program
{
  static void Foo (int x)    { Console.WriteLine ("1"); }
  static void Foo (string x) { Console.WriteLine ("2"); }
```

```
static void Main()
{
  dynamic x = 5;
  dynamic y = "watermelon";
```

```
    dynamic y = "watermelon";
    Foo (x);
    Foo (y);
  }
}
// 1
// 2
```

Because a dynamic receiver is not involved, the compiler can statically perform a basic check to see whether the dynamic call will succeed. It checks that a function with the right name and number of parameters exists. If no candidate is found, you get a compile-time error. For example:

```
class Program
```

```
class Program
{
  static void Foo (int x)    { Console.WriteLine ("1"); }
  static void Foo (string x) { Console.WriteLine ("2"); }
```

Advanced C#

```
static void Main()
{
```

```
  static void Main()
  {
    dynamic x = 5;
    Foo (x, x);
    Fook (x);
  }
}
```

```
// Compiler error - wrong number of parameters
// Compiler error - no such method name
```

Static Types in Dynamic Expressions

It's obvious that dynamic types are used in dynamic binding. It's not so obvious that

It's obvious that dynamic types are used in dynamic binding. It's not so obvious that static types are also used—wherever possible—in dynamic binding. Consider the following:

```
class Program
{
  static void Foo (object x, object y) { Console.WriteLine ("oo"); }
  static void Foo (object x, string y) { Console.WriteLine ("os"); }
  static void Foo (string x, object y) { Console.WriteLine ("so"); }
  static void Foo (string x, string y) { Console.WriteLine ("ss"); }

  static void Main()
  {
    object o = "hello";
    dynamic d = "goodbye";
    Foo (o, d);
  }
```

```
    Foo (o, d);
  }
}
// os
```


The call to `Foo(o,d)` is dynamically bound because one of its arguments, `d`, is `dynamic`. But since `o` is statically known, the binding—even though it occurs dynamically—will make use of that. In this case, overload resolution will pick the second implementation of `Foo` due to the static type of `o` and the runtime type of `d`. In other words, the compiler is "as static as it can possibly be."

Uncallable Functions

Some functions cannot be called dynamically. You cannot call:

- Extension methods (via extension method syntax)
- Any member of an interface
- Base members hidden by a subclass

Understanding why this is so is useful in understanding dynamic binding.

Dynamic binding requires two pieces of information: the name of the function to call, and the object upon which to call the function. However, in each of the three uncallable scenarios, an *additional type* is involved, which is known only at compile time. As of C# 4.0, there's no way to specify these additional types dynamically.

When calling extension methods, that additional type is implicit. It's the static class on which the extension method is defined. The compiler searches for it given the `using` directives in your source code. This makes extension methods compile-time-using directives in your source code. This makes extension methods compile-time-only concepts, since using directives melt away upon compilation (after they've done their job in the binding process in mapping simple names to namespace-qualified names).

When calling members via an interface, you specify that additional type via an implicit or explicit cast. There are two scenarios where you might want to do this: when calling explicitly implemented interface members, and when calling interface members implemented in a type internal to another assembly. We can illustrate the former with the following two types:

```
interface IFoo   { void Test();        }
class Foo : IFoo { void IFoo.Test() {} }
```

To call the Test method, we must cast to the IFoo interface. This is easy with static typing:

```
IFoo f = new Foo();   // Implicit cast to interface
f.Test();
```

Now consider the situation with dynamic typing:

```
IFoo f = new Foo();
```

```
IFoo f = new Foo();
dynamic d = f;
d.Test();                    // Exception thrown
```

The implicit cast shown in bold tells the *compiler* to bind subsequent member calls on `f` to `IFoo` rather than `Foo`—in other words, to view that object through the lens of the `IFoo` interface. However, that lens is lost at runtime, so the DLR cannot complete the binding. The loss is illustrated as follows:


```
Console.WriteLine (f.GetType().Name);

// Foo
```

A similar situation arises when calling a hidden base member: you must specify an additional type via either a cast or the `base` keyword—and that additional type is lost at runtime.