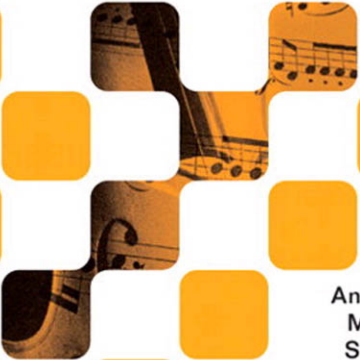
Special Annotated Edition for C# 3.0

The C# Programming Language

Third Edition



Microsoft .net Development Series

Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Peter Golde

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

The C# programming language / Anders Hejlsberg ... [et al.].—3rd ed.
p. cm.
Rev. ed of: The C# programming language / Anders Hejlsberg, Scott Wiltamuth, Peter Golde, 2nd ed. 2006.
Includes bibliographical references and index.
ISBN 978-0-321-56299-9 (alk. paper)
1. C# (Computer program language) I. Hejlsberg, Anders.

QA76.73.C154H45 2008
005.13'3—dc22

2008030025

ISBN-13: 978-0-321-56299-9
ISBN-10: 0-321-56299-2
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, October 2008

Foreword

It's been eight years since the launch of .NET in the summer of 2000. For me, the significance of .NET was the one–two combination of managed code for local execution and XML messaging for program-to-program communication. What wasn't obvious to me at the time was how important C# would be.

From the inception of .NET, C# has provided the primary means by which developers understand and interact with .NET. Ask the average .NET developer the difference between a value type and a reference type, and he or she will quickly say "struct versus class," not "types that derive from System.ValueType versus those that don't." Why? Because people use languages, not APIs, to communicate their ideas and intention to the runtime and, more importantly, to one another.

It's difficult to overstate how important having a great language has been to the success of the platform at large. C# was initially important to establish the baseline for how people think about .NET. It's proven even more important as .NET has evolved, and features such as iterators and true closures (also known as anonymous methods) have been introduced to developers as purely language features implemented by the C# compiler, not as features native to the platform. The fact that C# is a vital center of innovation for .NET became even more apparent with the release of C# 3.0, which introduced standardized query operators, compact lambda expressions, extension methods, and runtime access to expression trees—again, all driven out of the language and the compiler.

It's hard to talk about C# without also talking about its inventor and constant shepherd, Anders Hejlsberg. I had the distinct pleasure of participating in the recurring C# design meetings for a few months during the C# 3.0 design cycle, and it was enlightening watching Anders at work. His instincts for knowing what developers will and will not like is truly world class—yet at the same time, Anders is extremely inclusive of his design team and manages to get the best design possible.

With C# 3.0 in particular, Anders had an uncanny ability to take key ideas from the functional language community and make them accessible to a very broad audience. This is no trivial feat. Guy Steele once said of Java, "We were not out to win over the Lisp programmers; we were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp." When I look at C# 3.0, I know that C# has managed to drag at least one C++ developer (me) most of the rest of the way.

As good as C# is, people still need a document written in both natural language (English, in this case) and some formalism (BNF) to grok the subtleties and ensure that we're all speaking the same C#. The book you hold in your hands is that document. Based on my own experience, I can safely say that every .NET developer who reads it will have at least one "aha" moment and will be a better developer for it.

Enjoy.

Don Box
July 2008

Preface

The C# project started almost ten years ago, in December 1998, with the goal of creating a simple, modern, object-oriented, and type-safe programming language for the new and yet-to-be-named .NET platform. Since then, C# has come a long way. The language is now in use by more than one million programmers, and it has been released in three versions, each of which added several major new features.

This book, too, is in its third edition. A complete technical specification of the C# programming language, the third edition differs in several ways from the first two. Most notably, of course, it has been updated to cover all the new features of C# 3.0, including object and collection initializers, anonymous types, lambda expressions, query expressions, and partial methods. Most of these features are motivated by support for a more functional and declarative style of programming and, in particular, for Language Integrated Query (LINQ), which offers a unified approach to data querying across different kinds of data sources. LINQ, in turn, builds heavily on some of the features that were introduced in C# 2.0, including generics, iterators, and partial types.

Another change in the third edition is that the specification has been thoroughly reorganized. In the second edition of this book, the features introduced in C# 2.0 were described separately from the original C# 1.0 features. With a third helping of new features, this approach did not scale—the utility of the book would be impaired by the reader's need to correlate information from three different parts. Instead, the material is now organized by topic, with features from all three language versions presented together in an integrated manner.

A final but important departure from earlier editions is the inclusion of annotations in the text. We are very fortunate to be able to provide insightful guidance, background, and perspective from some of the world's leading experts in C# and .NET in the form of annotations throughout the book. We are very happy to see the annotations complement the core material and help the C# features spring to life.

Many people have been involved in the creation of the C# language. The language design team for C# 1.0 consisted of Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Peter Sollich, and Eric Gunnerson. For C# 2.0, the language design team consisted of Anders Hejlsberg, Peter Golde, Peter Hallam, Shon Katzenberger, Todd Proebsting, and Anson Horton. Furthermore, the design and implementation of generics in C# and the .NET Common Language Runtime are based on the "Gyro" prototype built by Don Syme and Andrew Kennedy of Microsoft Research. C# 3.0 was designed by Anders Hejlsberg, Peter Hallam, Shon Katzenberger, Dinesh Kulkarni, Erik Meijer, Mads Torgersen, and Matt Warren.

It is impossible to acknowledge the many people who have influenced the design of C#, but we are nonetheless grateful to all of them. Nothing good gets designed in a vacuum, and the constant feedback we receive from our large and enthusiastic community of developers is invaluable.

C# has been, and continues to be, one of the most challenging and exciting projects on which we've worked. We hope you enjoy using C# as much as we enjoyed creating it.

Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Seattle, Washington
July 2008

A compile-time error occurs if either of these requirements is not satisfied. Otherwise, the `&&` or `||` operation is evaluated by combining the user-defined `operator true` or `operator false` with the selected user-defined operator:

- The operation `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`, where `T.false(x)` is an invocation of the `operator false` declared in `T`, and `T.&(x, y)` is an invocation of the selected `operator &`. In other words, `x` is first evaluated, and `operator false` is invoked on the result to determine if `x` is definitely false. Then, if `x` is definitely false, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator &` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.
- The operation `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`, where `T.true(x)` is an invocation of the `operator true` declared in `T`, and `T.|(x, y)` is an invocation of the selected `operator |`. In other words, `x` is first evaluated, and `operator true` is invoked on the result to determine if `x` is definitely true. Then, if `x` is definitely true, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator |` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.

In either of these operations, the expression given by `x` is evaluated only once, and the expression given by `y` either is not evaluated or is evaluated exactly once.

For an example of a type that implements `operator true` and `operator false`, see §11.4.2.

7.12 The Null Coalescing Operator

The `??` operator is called the null coalescing operator.

null-coalescing-expression:
 conditional-or-expression
 conditional-or-expression `??` *null-coalescing-expression*

A null coalescing expression of the form `a ?? b` requires `a` to be of a nullable type or reference type. If `a` is non-null, the result of `a ?? b` is `a`; otherwise, the result is `b`. The operation evaluates `b` only if `a` is null.

The null coalescing operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ?? b ?? c` is evaluated as `a ?? (b ?? c)`. In general terms, an expression of the form E_1 `??` E_2 `??` … `??` E_N returns the first of the operands that is non-null, or `null` if all operands are null.

The type of the expression `a ?? b` depends on which implicit conversions are available between the types of the operands. In order of preference, the type of `a ?? b` is A_0, `A`, or `B`, where `A` is the type of `a`, `B` is the type of `b` (provided that `b` has a type), and A_0 is the underlying type of `A` if `A` is a nullable type, or `A` otherwise. Specifically, `a ?? b` is processed as follows:

- If `A` is not a nullable type or a reference type, a compile-time error occurs.
- If `A` is a nullable type and an implicit conversion exists from `b` to A_0, the result type is A_0. At runtime, `a` is evaluated first. If `a` is not null, `a` is unwrapped to type A_0, and this becomes the result. Otherwise, `b` is evaluated and converted to type A_0, and this becomes the result.
- Otherwise, if an implicit conversion exists from `b` to `A`, the result type is `A`. At runtime, `a` is evaluated first. If `a` is not null, `a` becomes the result. Otherwise, `b` is evaluated and converted to type `A`, and this becomes the result.
- Otherwise, if `b` has a type `B` and an implicit conversion exists from A_0 to `B`, the result type is `B`. At runtime, `a` is evaluated first. If `a` is not null, `a` is unwrapped to type A_0 (unless `A` and A_0 are the same type) and converted to type `B`, and this becomes the result. Otherwise, `b` is evaluated and becomes the result.
- Otherwise, `a` and `b` are incompatible, and a compile-time error occurs.

ERIC LIPPERT These conversion rules considerably complicate the transformation of a null coalescing operator into an expression tree. In some cases, the compiler must emit an additional expression tree lambda specifically to handle the conversion logic.

CHRIS SELLS The `??` operator is useful for setting default values for reference types or nullable value types. For example,

```
Foo f1 = ...;
Foo f2 = f1 ?? new Foo(...);
int?i1 = ...;
int i2 = i1 ?? 452;
```

FRITZ ONION When I first saw the new `??` operator in C# 2.0, I was a bit skeptical. Did we really need another terse operator to further obfuscate our code? Over the last year or two, however, I've actually found myself using it quite a bit, and I think this operator actually enhances readability once you get used to it. So I guess the answer was yes, we—or I, at least—did need the `??` operator.

Consider the example of implementing a property that is backed by `ViewState` in ASP.NET (say, for a custom control):

```
 public string Title
{
  get { return (string)ViewState["title"] ?? "Default title"; }
  set { ViewState["title"] = value; }
}
```

I use this pattern quite a bit when implementing reference type properties backed by `ViewState`, and I quite like the way it succinctly expresses the check for `null`.

7.13 Conditional Operator

The `?:` operator is called the conditional operator (or sometimes the ternary operator).

conditional-expression:
  null-coalescing-expression
  null-coalescing-expression `?` *expression* `:` *expression*

A conditional expression of the form `b ? x : y` first evaluates the condition `b`. Then, if `b` is `true`, `x` is evaluated and becomes the result of the operation. Otherwise, `y` is evaluated and becomes the result of the operation. A conditional expression never evaluates both `x` and `y`.

DON BOX Having learned Lisp late in life, I almost never used the conditional operator when I was a C++ programmer. When I started working with C# 3.0, its value became obvious. I use the conditional operator a fair amount now, and not just inside of lambda expressions.

The conditional operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`.

The first operand of the `?:` operator must be an expression of a type that can be implicitly converted to `bool`, or an expression of a type that implements `operator true`. If neither of these requirements is satisfied, a compile-time error occurs.

The second and third operands of the `?:` operator control the type of the conditional expression. Let `X` and `Y` be the types of the second and third operands. Then,

- If `X` and `Y` are the same type, then this is the type of the conditional expression.

- Otherwise, if an implicit conversion (§6.1) exists from `X` to `Y`, but not from `Y` to `X`, then `Y` is the type of the conditional expression.
- Otherwise, if an implicit conversion (§6.1) exists from `Y` to `X`, but not from `X` to `Y`, then `X` is the type of the conditional expression.
- Otherwise, no expression type can be determined, and a compile-time error occurs.

> **ERIC LIPPERT** The Microsoft C# compiler actually implements a slightly different algorithm: It checks for conversions from the *expressions* to the types, not from *types* to types. In most cases, the difference does not matter and it would break existing code to change it now.

The runtime processing of a conditional expression of the form `b ? x : y` consists of the following steps:

- First, `b` is evaluated, and the `bool` value of `b` is determined:
  - If an implicit conversion from the type of `b` to `bool` exists, then this implicit conversion is performed to produce a `bool` value.
  - Otherwise, the `operator true` defined by the type of `b` is invoked to produce a `bool` value.
- If the `bool` value produced by the previous step is `true`, then `x` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.
- Otherwise, `y` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.

7.14 Anonymous Function Expressions

An ***anonymous function*** is an expression that represents an "in-line" method definition. An anonymous function does not have a value in and of itself, but rather is convertible to a compatible delegate or expression tree type. The evaluation of an anonymous function conversion depends on the target type of the conversion: If it is a delegate type, the conversion evaluates to a delegate value referencing the method that the anonymous function defines. If it is an expression tree type, the conversion evaluates to an expression tree that represents the structure of the method as an object structure.

> **DON BOX** Of all of the recently introduced features in C#, anonymous functions have been the one that has increased my productivity the most. Specifically, not having to factor blocks of functionality into named classes and methods makes it extremely efficient for me to get an idea from my brain into executable code.

For historical reasons, two syntactic flavors of anonymous functions exist—namely, *lambda-expression*s and *anonymous-method-expression*s. For almost all purposes, *lambda-expression*s are more concise and expressive than *anonymous-method-expression*s, which remain in the language for backward compatibility.

lambda-expression:
  anonymous-function-signature `=>` *anonymous-function-body*

anonymous-method-expression:
  `delegate` *explicit-anonymous-function-signature*$_{opt}$ *block*

anonymous-function-signature:
  explicit-anonymous-function-signature
  implicit-anonymous-function-signature

explicit-anonymous-function-signature:
  `(` *explicit-anonymous-function-parameter-list*$_{opt}$ `)`

explicit-anonymous-function-parameter-list
  explicit-anonymous-function-parameter
  explicit-anonymous-function-parameter-list `,` *explicit-anonymous-function-parameter*

explicit-anonymous-function-parameter:
  anonymous-function-parameter-modifier$_{opt}$ *type* *identifier*

anonymous-function-parameter-modifier:
  `ref`
  `out`

implicit-anonymous-function-signature:
  `(` *implicit-anonymous-function-parameter-list*$_{opt}$ `)`
  implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list
  implicit-anonymous-function-parameter
  implicit-anonymous-function-parameter-list `,` *implicit-anonymous-function-parameter*

implicit-anonymous-function-parameter:
  identifier

anonymous-function-body:
  expression
  block

The `=>` operator has the same precedence as assignment (`=`) and is right-associative.

The parameters of an anonymous function in the form of a *lambda-expression* can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the anonymous function occurs—specifically, when the anonymous function is converted to a compatible delegate type or expression tree type, that type provides the parameter types (§6.5).

> **BILL WAGNER** In general, your anonymous functions will be more resilient if you rely on implicit typing.

In an anonymous function with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, an anonymous function of the form

`(` *param* `) =>` *expr*

can be abbreviated to

param `=>` *expr*

The parameter list of an anonymous function in the form of an *anonymous-method-expression* is optional. If given, the parameters must be explicitly typed. If not, the anonymous function is convertible to a delegate with any parameter list not containing `out` parameters.

Some examples of anonymous functions follow:

```
x => x + 1                        // Implicitly typed, expression body

x => { return x + 1; }            // Implicitly typed, statement body

(int x) => x + 1                  // Explicitly typed, expression body

(int x) => { return x + 1; }      // Explicitly typed, statement body

(x, y) => x * y                   // Multiple parameters

() => Console.WriteLine()         // No parameters

delegate (int x) { return x + 1; } // Anonymous method expression

delegate { return 1 + 1; }        // Parameter list omitted
```

The behavior of *lambda-expression*s and *anonymous-method-expression*s is the same except for the following points:

- *anonymous-method-expression*s permit the parameter list to be omitted entirely, yielding convertibility to delegate types of any list of value parameters.
- *lambda-expression*s permit parameter types to be omitted and inferred, whereas *anonymous-method-expression*s require parameter types to be explicitly stated.
- The body of a *lambda-expression* can be an expression or a statement block, whereas the body of an *anonymous-method-expression* must be a statement block.
- Because only *lambda-expression*s can have *expression* bodies, no *anonymous-method-expression* can be successfully converted to an expression tree type (§4.6).

> **BILL WAGNER** This point is important for building queries that rely on expression trees, such as those in Linq to SQL and Linq to Entities.

7.14.1 Anonymous Function Signatures

The optional *anonymous-function-signature* of an anonymous function defines the names and optionally the types of the formal parameters for the anonymous function. The scope of the parameters of the anonymous function is the *anonymous-function-body* (§3.7). Together with the parameter list (if given), the *anonymous-method-body* constitutes a declaration space (§3.3). For this reason, it is a compile-time error for the name of a parameter of the anonymous function to match the name of a local variable, local constant, or parameter whose scope includes the *anonymous-method-expression* or *lambda-expression*.

If an anonymous function has an *explicit-anonymous-function-signature*, then the set of compatible delegate types and expression tree types is restricted to those that have the same parameter types and modifiers in the same order. In contrast to method group conversions (§6.6), contravariance of anonymous function parameter types is not supported. If an anonymous function does not have an *anonymous-function-signature*, then the set of compatible delegate types and expression tree types is restricted to those that have no `out` parameters.

An *anonymous-function-signature* cannot include attributes or a parameter array. Nevertheless, an *anonymous-function-signature* may be compatible with a delegate type whose parameter list contains a parameter array.

Note that conversion to an expression tree type, even if compatible, may still fail at compile time (§4.6).

> **ERIC LIPPERT** This is a subtle point. If you have two overloads—say, `void M(Expression<Func<Giraffe>> f)` and `void M(Func<Animal> f)`—and a call `M(()=>myGiraffes[++i])`, then the expression tree overload is chosen as the better overload. In this situation, a compile-time error occurs because the increment operator is illegal inside an expression tree.

7.14.2 Anonymous Function Bodies

The body (*expression* or *block*) of an anonymous function is subject to the following rules:

- If the anonymous function includes a signature, the parameters specified in the signature are available in the body. If the anonymous function has no signature, it can be converted to a delegate type or expression type having parameters (§6.5), but the parameters cannot be accessed in the body.
- Except for `ref` or `out` parameters specified in the signature (if any) of the nearest enclosing anonymous function, it is a compile-time error for the body to access a `ref` or `out` parameter.
- When the type of `this` is a struct type, it is a compile-time error for the body to access `this`. This is true whether the access is explicit (as in `this.x`) or implicit (as in `x` where `x` is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup returns a member of the struct.
- The body has access to the outer variables (§7.14.4) of the anonymous function. Access of an outer variable will reference the instance of the variable that is active at the time the *lambda-expression* or *anonymous-method-expression* is evaluated (§7.14.5).
- It is a compile-time error for the body to contain a `goto` statement, `break` statement, or `continue` statement whose target is outside the body or within the body of a contained anonymous function.
- A `return` statement in the body returns control from an invocation of the nearest enclosing anonymous function, not from the enclosing function member. An expression specified in a `return` statement must be compatible with the delegate type or expression tree type to which the nearest enclosing *lambda-expression* or *anonymous-method-expression* is converted (§6.5).

It is explicitly unspecified whether there is any way to execute the block of an anonymous function other than through evaluation and invocation of the *lambda-expression* or *anonymous-method-expression*. In particular, the compiler may choose to implement an anonymous function by synthesizing one or more named methods or types. The names of any such synthesized elements must be of a form reserved for compiler use.

7.14.3 Overload Resolution

Anonymous functions in an argument list participate in type inference and overload resolution. Refer to §7.4.2.3 for the exact rules governing their behavior.

The following example illustrates the effect of anonymous functions on overload resolution.

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

The `ItemList<T>` class has two `Sum` methods. Each takes a `selector` argument, which extracts the value to sum over from a list item. The extracted value can be either an `int` or a `double`, and the resulting sum is likewise either an `int` or a `double`.

The `Sum` methods could, for example, be used to compute sums from a list of detail lines in some order.

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}

void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

In the first invocation of `orderDetails.Sum`, both `Sum` methods are applicable because the anonymous function `d => d.UnitCount` is compatible with both `Func<Detail,int>` and `Func<Detail,double>`. However, overload resolution picks the first `Sum` method because the conversion to `Func<Detail,int>` is better than the conversion to `Func<Detail,double>`.

In the second invocation of `orderDetails.Sum`, only the second `Sum` method is applicable because the anonymous function `d => d.UnitPrice * d.UnitCount` produces a value of type `double`. Thus overload resolution picks the second `Sum` method for that invocation.

7.14.4 Outer Variables

Any local variable, value parameter, or parameter array whose scope includes the *lambda-expression* or *anonymous-method-expression* is called an ***outer variable*** of the anonymous function. In an instance function member of a class, the `this` value is considered a value parameter and is an outer variable of any anonymous function contained within the function member.

> **BILL WAGNER** This is the formal definition of how closures are implemented in C#. It's a great addition.

7.14.4.1 *Captured Outer Variables*

When an outer variable is referenced by an anonymous function, the outer variable is said to have been ***captured*** by the anonymous function. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated (§5.1.7). However, the lifetime of a captured outer variable is extended at least until the delegate or expression tree created from the anonymous function becomes eligible for garbage collection.

> **BILL WAGNER** In the next example, notice that `x` has a longer life than you would expect, because it is captured by the anonymous method result. If `x` were an expensive resource, that behavior should be avoided by limiting the lifetime of the anonymous method.

In the example

```
using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }
```

```
    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}
```

the local variable `x` is captured by the anonymous function, and the lifetime of `x` is extended at least until the delegate returned from `F` becomes eligible for garbage collection (which doesn't happen until the very end of the program). Because each invocation of the anonymous function operates on the same instance of `x`, the example produces the following output:

```
1
2
3
```

When a local variable or a value parameter is captured by an anonymous function, the local variable or parameter is no longer considered to be a fixed variable (§18.3), but is instead considered to be a moveable variable. Thus any `unsafe` code that takes the address of a captured outer variable must first use the `fixed` statement to fix the variable.

7.14.4.2 *Instantiation of Local Variables*

A local variable is considered to be ***instantiated*** when execution enters the scope of the variable. For example, when the following method is invoked, the local variable `x` is instantiated and initialized three times—once for each iteration of the loop.

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

By comparison, moving the declaration of `x` outside the loop results in a single instantiation of `x`:

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

When not captured, there is no way to observe exactly how often a local variable is instantiated—because the lifetimes of the instantiations are disjoint, it is possible for each instantiation to simply use the same storage location. However, when an anonymous function captures a local variable, the effects of instantiation become apparent.

The example

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

produces the following output:

```
1
3
5
```


When the declaration of x is moved outside the loop,

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

the output changes as follows:

```
5
5
5
```

If a *for-statement* declares an iteration variable, that variable itself is considered to be declared outside of the loop. Thus, if the example is changed to capture the iteration variable itself,

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

only one instance of the iteration variable is captured, which produces the following output:

```
3
3
3
```

It is possible for anonymous function delegates to share some captured variables, yet have separate instances of others. For example, if `F` is changed to

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}
```

the three delegates capture the same instance of `x` but separate instances of `y`, and the output is as follows:

```
1 1
2 1
3 1
```

Separate anonymous functions can capture the same instance of an outer variable. In the example

```
using System;

delegate void Setter(int value);

delegate int Getter();

class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

the two anonymous functions capture the same instance of the local variable `x`, and they can "communicate" through that variable. This example results in the following output:

```
5
10
```

7.14.5 Evaluation of Anonymous Function Expressions

An anonymous function `F` must always be converted to a delegate type `D` or an expression tree type `E`, either directly or through the execution of a delegate creation expression `new D(F)`. This conversion determines the result of the anonymous function, as described in §6.5.

7.15 Query Expressions

Query expressions provide a language-integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

query-expression:
  from-clause query-body

from-clause:
  `from` *type*$_{opt}$ *identifier* `in` *expression*

query-body:
  query-body-clauses$_{opt}$ *select-or-group-clause query-continuation*$_{opt}$

query-body-clauses:
  query-body-clause
  query-body-clauses query-body-clause

query-body-clause:
  from-clause
  let-clause
  where-clause
  join-clause
  join-into-clause
  orderby-clause

let-clause:
  `let` *identifier* `=` *expression*

where-clause:
  `where` *boolean-expression*

join-clause:
`join` *type*$_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression*

join-into-clause:
`join` *type*$_{opt}$ *identifier* `in` *expression* `on` *expression* `equals` *expression* `into` *identifier*

orderby-clause:
`orderby` *orderings*

orderings:
ordering
orderings `,` *ordering*

ordering:
expression *ordering-direction*$_{opt}$

ordering-direction:
`ascending`
`descending`

select-or-group-clause:
select-clause
group-clause

select-clause:
`select` *expression*

group-clause:
`group` *expression* `by` *expression*

query-continuation:
`into` *identifier* *query-body*

A query expression begins with a `from` clause and ends with either a `select` or `group` clause. The initial `from` clause can be followed by zero or more `from`, `let`, `where`, `join`, or `orderby` clauses. Each `from` clause is a generator introducing a ***range variable***, which ranges over the elements of a ***sequence***. Each `let` clause introduces a range variable representing a value computed by means of previous range variables. Each `where` clause is a filter that excludes items from the result. Each `join` clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each `orderby` clause reorders items according to specified criteria. The final `select` or `group` clause specifies the shape of the result in terms of the range variables. Finally, an `into` clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

7.15.1 Ambiguities in Query Expressions

Query expressions contain a number of "contextual keywords"—that is, identifiers that have special meaning in a given context. Specifically, these contextual keywords are `from`, `where`, `join`, `on`, `equals`, `into`, `let`, `orderby`, `ascending`, `descending`, `select`, `group`, and `by`. To avoid ambiguities in query expressions caused by mixed use of these identifiers as keywords and simple names, the identifiers are always considered keywords when they occur anywhere within a query expression.

For this purpose, a query expression is any expression that starts with "`from` *identifier*" followed by any token except "`;`", "`=`", or "`,`".

To use these words as identifiers within a query expression, prefix them with "`@`" (§2.4.2).

7.15.2 Query Expression Translation

The C# language does not directly specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the query expression pattern (§7.15.3). Specifically, query expressions are translated into invocations of methods named `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, and `Cast`. These methods are expected to have particular signatures and result types, as described in §7.15.3. They can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any type binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but it is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations. This processing may, in turn, uncover errors—for example, if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

> **BILL WAGNER** This entire section is a great way to understand how query expressions are translated into method calls and possibly extension method calls.

A query expression is processed by repeatedly applying the following translations until no further reductions are possible. The translations are listed in order of application: Each section assumes that the translations in the preceding sections have been performed exhaustively, and once exhausted, a section will not be revisited later in the processing of the same query expression.

Assignment to range variables is not allowed in query expressions. However, a C# implementation is permitted to not always enforce this restriction, because satisfying this constraint may sometimes not be possible with the syntactic translation scheme presented here.

Certain translations inject range variables with *transparent identifiers* denoted by *. The special properties of transparent identifiers are discussed further in §7.15.2.7.

> **CHRIS SELLS** As much as I like the C# 3.0 query syntax, sometimes it's difficult to keep the translations in my head. Don't feel bad if you occasionally feel the need to write out your queries using the method call syntax. Also, any query methods that you implement yourself will not have language constructs, so sometimes you won't have any choice except to use the method call syntax.

7.15.2.1 *select and groupby Clauses with Continuations*

A query expression with a continuation

```
from … into x …
```

is translated into

```
from x in ( from … ) …
```

The translations in the following sections assume that queries have no `into` continuations.

The example

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

is translated into

```
from g in
    from c in customers
    group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

Its final translation is

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

> **JOSEPH ALBAHARI** The purpose of a query continuation is to allow further clauses after a `select` or `group` clause (which would otherwise terminate the query). After a query continuation, the former range variable, and any variables that were introduced through `join` or `let` clauses, are out of scope. In contrast, a `let` clause acts like a nondestructive `select`: It keeps the former range variable, and other query variables, in scope.
>
> The identifier introduced by a query continuation can be the same as the preceding range variable.

7.15.2.2 *Explicit Range Variable Types*

A `from` clause that explicitly specifies a range variable type

```
from T x in e
```

is translated into

```
from x in ( e ) . Cast < T > ( )
```

A `join` clause that explicitly specifies a range variable type

```
join T x in e on k1 equals k2
```

is translated into

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

The translations in the following sections assume that queries have no explicit range variable types.

The example

```
from Customer c in customers
where c.City == "London"
select c
```

is translated into

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

The final translation is

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Explicit range variable types are useful for querying collections that implement the non-generic `IEnumerable` interface, but not the generic `IEnumerable<T>` interface. In the preceding example, this would be the case if `customers` were of type `ArrayList`.

7.15.2.3 *Degenerate Query Expressions*

A query expression of the form

```
from x in e select x
```

is translated into

```
( e ) . Select ( x => x )
```

The example

```
from c in customers
select c
```

is translated into

```
customers.Select(c => c)
```

A degenerate query expression is one that trivially selects the elements of the source. A later phase of the translation removes degenerate queries introduced by other translation steps by replacing those queries with their source. In this situation, it is important to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. As a consequence, this step protects degenerate queries written directly in source code by explicitly calling `Select` on the source. It is then up to the implementers of `Select` and other query operators to ensure that these methods never return the source object itself.

7.15.2.4 *from, let, where, join, and orderby Clauses*

JOSEPH ALBAHARI The cumbersome-looking translations in this section are what make query syntax really useful: They eliminate the need to write out cumbersome queries by hand. Without this problem, there might have been little justification for introducing query expression syntax into C# 3.0, given the capabilities of lambda expressions and extension methods.

The common theme in the more complex translations is the process of projecting into a temporary anonymous type so as to keep the former range variable in scope following a `let`, `from`, or `join` clause.

A query expression with a second `from` clause followed by a `select` clause

```
from x1 in e1
from x2 in e2
select v
```

is translated into

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

A query expression with a second `from` clause followed by something other than a `select` clause

```
from x1 in e1
from x2 in e2
...
```

is translated into

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
...
```

A query expression with a `let` clause

```
from x in e
let y = f
...
```

is translated into

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

A query expression with a `where` clause

```
from x in e
where f
...
```

is translated into

```
from x in ( e ) . Where ( x => f )
...
```

A query expression with a `join` clause without an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

is translated into

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

A query expression with a `join` clause without an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

is translated into

```
from * in ( e1 ) . Join(
    e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```

A query expression with a `join` clause with an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

is translated into

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

A query expression with a `join` clause with an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

is translated into

```
from * in ( e1 ) . GroupJoin(
    e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
...
```

A query expression with an `orderby` clause

```
from x in e
orderby k1 , k2 , ... , kn
...
```

is translated into

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
 ... .
ThenBy ( x => kn )
...
```

If an ordering clause specifies a `descending` direction indicator, an invocation of `OrderByDescending` or `ThenByDescending` is produced instead.

The following translations assume that there are no `let`, `where`, `join`, or `orderby` clauses, and no more than the one initial `from` clause in each query expression.

The example

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
customers.
SelectMany(c => c.Orders,
     (c,o) => new { c.Name, o.OrderID, o.Total }
)
```

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

The final translation is

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

where `x` is a compiler-generated identifier that is otherwise invisible and inaccessible.

The example

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

is translated into

```
from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

The final translation is

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

where x is a compiler-generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

is translated into

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

is translated into

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

The final translation is

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
    (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

where `x` and `y` are compiler-generated identifiers that are otherwise invisible and inaccessible.

The example

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

has the final translation

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

7.15.2.5 *select Clauses*

A query expression of the form

```
from x in e select v
```

is translated into

```
( e ) . Select ( x => v )
```

except when v is the identifier x. In the latter case, the translation is simply

```
( e )
```

For example,

```
from c in customers.Where(c => c.City == "London")
select c
```

is simply translated into

```
customers.Where(c => c.City == "London")
```

7.15.2.6 *groupby Clauses*

A query expression of the form

```
from x in e group v by k
```

is translated into

```
( e ) . GroupBy ( x => k , x => v )
```

except when v is the identifier x. In the latter case, the translation is

```
( e ) . GroupBy ( x => k )
```

The example

```
from c in customers
group c.Name by c.Country
```

is translated into

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

7.15.2.7 *Transparent Identifiers*

Certain translations inject range variables with ***transparent identifiers*** denoted by *. Transparent identifiers are not a proper language feature; they exist only as an intermediate step in the query expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into anonymous functions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

- When a transparent identifier occurs as a parameter in an anonymous function, the members of the associated anonymous type are automatically in scope in the body of the anonymous function.
- When a member with a transparent identifier is in scope, the members of that member are in scope as well.
- When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.

In the translation steps described earlier, transparent identifiers are always introduced together with anonymous types, with the intent of capturing multiple range variables as members of a single object. An implementation of C# is permitted to use a different mechanism than anonymous types to group together multiple range variables. The following translation examples assume that anonymous types are used, and show how transparent identifiers can be translated away.

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

is translated into

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

which is further translated into

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

When transparent identifiers are erased, the final translation is equivalent to

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

where x is a compiler-generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

is translated into

```
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

which is further reduced to

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

The final translation is

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
    (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
    (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

where x, y, and z are compiler-generated identifiers that are otherwise invisible and inaccessible.

MADS TORGERSEN The last example demonstrates one of the most powerful aspects of query expressions—the ability to introduce multiple range variables and have them pass through subsequent query operators in a manner transparent to the programmer.

7.15.3 The Query Expression Pattern

The ***query expression pattern*** establishes a pattern of methods that types can implement to support query expressions. Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement the query expression pattern. For example, the methods of the pattern can be implemented as instance methods or as extension methods because both kinds of methods have the same invocation syntax. Likewise, the methods can request delegates or expression trees because anonymous functions are convertible to both.

The recommended shape of a generic type C<T> that supports the query expression pattern is shown below. A generic type is used to illustrate the proper relationships between parameter and result types, but it is possible to implement the pattern for nongeneric types as well.

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);

    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);
```

7. Expressions

```
    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

These methods use the generic delegate types `Func<T1, R>` and `Func<T1, T2, R>`, but they could equally well have used other delegate or expression tree types with the same relationships in parameter and result types.

Notice the recommended relationship between `C<T>` and `O<T>`, which ensures that the `ThenBy` and `ThenByDescending` methods are available only on the result of an `OrderBy` or `OrderByDescending`. Also notice the recommended shape of the result of `GroupBy`—a sequence of sequences, where each inner sequence has an additional `Key` property.

> **BILL WAGNER** `ThenBy` will often have better performance than `OrderBy`, because it needs to sort only inner sequences that have more than one value.

The `System.Linq` namespace provides an implementation of the query operator pattern for any type that implements the `System.Collections.Generic.IEnumerable<T>` interface.

> **BILL WAGNER** There is also an implementation for any type that implements `IQueryable<T>`.

> **ERIC LIPPERT** This signature for `Join` is one of the primary motivators of the "accumulate bounds and then fix to the best one" part of the method type inference algorithm. If the inner key is, say, of type `int`, and the outer key of of type `int?`, then rather than having type inference fail due to the "contradiction," it is better to simply pick the more general of the two types. Because every `int` is an `int?`, the type inference algorithm would choose `int?` for `K`.

7.16 Assignment Operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

assignment:
  unary-expression assignment-operator expression

assignment-operator:
  `=`
  `+=`
  `-=`
  `*=`
  `/=`
  `%=`
  `&=`
  `|=`
  `^=`
  `<<=`
  right-shift-assignment

The left operand of an assignment must be an expression classified as a variable, a property access, an indexer access, or an event access.

The `=` operator is called the ***simple assignment operator***. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator may not be an event access (except as described in §10.8.1). The simple assignment operator is described in §7.16.1.

The assignment operators other than the `=` operator are called ***compound assignment operators***. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in §7.16.2.

The `+=` and `-=` operators with an event access expression as the left operand are called ***event assignment operators***. No other assignment operator is valid with an event access as the left operand. The event assignment operators are described in §7.16.3.

The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a = b = c` is evaluated as `a = (b = c)`.

7.16.1 Simple Assignment

The = operator is called the simple assignment operator. In a simple assignment, the right operand must be an expression of a type that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer must have a `set` accessor. If this is not the case, a compile-time error occurs.

The runtime processing of a simple assignment of the form `x = y` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (§6.1).
  - If the variable given by `x` is an array element of a *reference-type*, a runtime check is performed to ensure that the value computed for `y` is compatible with the array instance of which `x` is an element. The check succeeds if `y` is `null`, or if an implicit reference conversion (§6.1.6) exists from the actual type of the instance referenced by `y` to the actual element type of the array instance containing `x`. Otherwise, a `System.ArrayTypeMismatchException` is thrown.
  - The value resulting from the evaluation and conversion of `y` is stored into the location given by the evaluation of `x`.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `set` accessor invocation.
  - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (§6.1).
  - The `set` accessor of `x` is invoked with the value computed for `y` as its `value` argument.

The array covariance rules (§12.5) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, assignment to an array element of a *reference-type* requires a runtime check to ensure that the value being assigned is compatible with the array instance. In the example

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;                   // Okay
oa[1] = "Hello";                // Okay
oa[2] = new ArrayList();        // ArrayTypeMismatchException
```

the last assignment causes a `System.ArrayTypeMismatchException` to be thrown because an instance of `ArrayList` cannot be stored in an element of a `string[]`.

> **BILL WAGNER** This point implies that array assignment does not copy the array, but rather adds a new reference to the same storage.

When a property or indexer declared in a *struct-type* is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. Because of the points raised in §7.5.4, the same rule also applies to fields.

Given the declarations:

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int X {
        get { return x; }
        set { x = value; }
    }

    public int Y {
        get { return y; }
        set { y = value; }
    }
}

struct Rectangle
{
    Point a, b;

    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }
```

```
        public Point A {
            get { return a; }
            set { a = value; }
        }

        public Point B {
            get { return b; }
            set { b = value; }
        }
    }
```

in the example

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the example

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

the assignments are all invalid, because `r.A` and `r.B` are not variables.

> **JOSEPH ALBAHARI** An early release of the C# 1.0 compiler allowed assignments such as `r.A.X = 10`—but they failed silently because `r.A` returns a *copy* of a `Point` (i.e., a value) rather than a variable. People found this behavior confusing, so the condition was detected and reported as an error.

> **BILL WAGNER** This discussion highlights yet another reason why structs should be immutable.

7.16.2 Compound Assignment

An operation of the form `x` *op*= `y` is processed by applying binary operator overload resolution (§7.2.4) as if the operation was written `x` *op* `y`. Then,

- If the return type of the selected operator is *implicitly* convertible to the type of `x`, the operation is evaluated as `x = x` *op* `y`, except that `x` is evaluated only once.

- Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is *explicitly* convertible to the type of `x`, and if `y` is *implicitly* convertible to the type of `x` or the operator is a shift operator, then the operation is evaluated as `x = (T)(x` *op* `y)`, where `T` is the type of `x`, except that `x` is evaluated only once.
- Otherwise, the compound assignment is invalid, and a compile-time error occurs.

The term "evaluated only once" means that in the evaluation of `x` *op* `y`, the results of any constituent expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. For example, in the assignment `A()[B()] += C()`, where `A` is a method returning `int[]`, and `B` and `C` are methods returning `int`, the methods are invoked only once, in the order `A`, `B`, `C`.

When the left operand of a compound assignment is a property access or indexer access, the property or indexer must have both a `get` accessor and a `set` accessor. If this is not the case, a compile-time error occurs.

The second rule permits `x` *op*`= y` to be evaluated as `x = (T)(x` *op* `y)` in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type `sbyte`, `byte`, `short`, `ushort`, or `char`. Even when both arguments are of one of those types, the predefined operators produce a result of type `int`, as described in §7.2.6.2. Thus, without a cast, it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that `x` *op*`= y` is permitted if both of `x` *op* `y` and `x = y` are permitted. In the example

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;                 // Okay
b += 1000;              // Error, b = 1000 not permitted
b += i;                 // Error, b = i not permitted
b += (byte)i;           // Okay

ch += 1;                // Error, ch = 1 not permitted
ch += (char)1;          // Okay
```

the intuitive reason for each error is that a corresponding simple assignment would also have been an error.

This also means that compound assignment operations support lifted operations. In the example

```
int? i = 0;
i += 1;            // Okay
```

the lifted operator `+(int?,int?)` is used.

7.16.3 Event Assignment

If the left operand of a `+=` or `-=` operator is classified as an event access, then the expression is evaluated as follows:

- The instance expression, if any, of the event access is evaluated.
- The right operand of the `+=` or `-=` operator is evaluated and, if required, converted to the type of the left operand through an implicit conversion (§6.1).
- An event accessor of the event is invoked, with an argument list consisting of the right operand, after evaluation and, if necessary, conversion. If the operator was `+=`, the `add` accessor is invoked; if the operator was `-=`, the `remove` accessor is invoked.

An event assignment expression does not yield a value. Thus an event assignment expression is valid only in the context of a *statement-expression* (§8.6).

7.17 Expressions

An *expression* is either a *non-assignment-expression* or an *assignment*.

expression:
  non-assignment-expression
  assignment

non-assignment-expression:
  conditional-expression
  lambda-expression
  query-expression

7.18 Constant Expressions

A *constant-expression* is an expression that can be fully evaluated at compile time.

constant-expression:
  expression

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal)
- References to `const` members of class and struct types

- References to members of enumeration types
- References to `const` parameters or local variables
- Parenthesized subexpressions, which are themselves constant expressions
- Cast expressions, provided the target type is one of the types listed above
- `checked` and `unchecked` expressions
- Default value expressions
- The predefined `+`, `–`, `!`, and `~` unary operators
- The predefined `+`, `–`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above
- The `?:` conditional operator

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the `null` value

Other conversions including boxing, unboxing, and implicit reference conversions of non-null values are not permitted in constant expressions. For example,

```
class C {
    const object i = 5;         // Error: boxing conversion not permitted
    const object str = "hello"; // Error: implicit reference conversion
}
```

In this example, the initialization of `i` is an error because a boxing conversion is required. The initialization of `str` is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile time. This is true even if the expression is a subexpression of a larger expression that contains nonconstant constructs.

The compile-time evaluation of constant expressions uses the same rules as runtime evaluation of nonconstant expressions, except that where runtime evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (§7.18).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile time.

- Constant declarations (§10.4)
- Enumeration member declarations (§14.3)
- `case` labels of a `switch` statement (§8.7.2)
- `goto case` statements (§8.9.3)
- Dimension lengths in an array creation expression (§7.5.10.4) that includes an initializer
- Attributes (§17)

An implicit constant expression conversion (§6.1.8) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

7.19 Boolean Expressions

A *boolean-expression* is an expression that yields a result of type `bool`, either directly or through application of `operator true` in certain contexts as specified in the following discussion.

boolean-expression:
  expression

The controlling conditional expression of an *if-statement* (§8.7.1), *while-statement* (§8.8.1), *do-statement* (§8.8.2), or *for-statement* (§8.8.3) is a *boolean-expression*. The controlling conditional expression of the `?:` operator (§7.13) follows the same rules as a *boolean-expression*, but for reasons of operator precedence is classified as a *conditional-or-expression*.

A *boolean-expression* is required to be of a type that can be implicitly converted to `bool` or of a type that implements `operator true`. If neither requirement is satisfied, a compile-time error occurs.

When a boolean expression is of a type that cannot be implicitly converted to `bool` but does implement `operator true`, then following evaluation of the expression, the `operator true` implementation provided by that type is invoked to produce a `bool` value.

The `DBBool` struct type in §11.4.2 provides an example of a type that implements `operator true` and `operator false`.

Index

B

D

E

F

G

J

K

L

M

N

O

P

Q

R

T

V