C# 3.0

1. Implicitly Typed Local Variables

2. Implicitly Typed Arrays

3. Anonymous Types

4. Object and Collection Initializers

5. Extension Methods

6. Lambda Expressions

7. Expression Trees

8. Automatically Implemented Properties

9. Query Expressions (LINQ)

10. Partial Classes and Methods (Partial Classes, Structs, and Interfaces were available in C# 2.0)

**1. Implicitly Typed Local Variables**

Local variables can be given an inferred "type" of **var** instead of an explicit type. The **var** keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET Framework class library.

**2. Implicitly Typed Arrays**

You can create an implicitly-typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly-typed variable also apply to implicitly-typed arrays.

Implicitly-typed arrays are usually used in query expressions together with anonymous types and object and collection initializers.

**3. Anonymous Types**

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You create anonymous types by using the [new](http://msdn.microsoft.com/en-us/library/51y09td4.aspx) operator together with an object initializer

**4. Object and Collection Initializers**

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to explicitly invoke a constructor. The following example shows how to use an object initializer with a named type, Cat. Note the use of auto-implemented properties in the Cat class.

**5. Extension Methods (C# Programming Guide)**

Updated: October 2010

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

**6. Lambda Expressions**

A *lambda expression* is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.

All lambda expressions use the lambda operator [=>](http://msdn.microsoft.com/en-us/library/bb311046.aspx), which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block. The lambda expression x => x \* x is read "x goes to x times x."

**7. Expression Trees**

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as x < y.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries.

**8. Auto-Implemented Properties (C# Programming Guide)**

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's get and set accessors.

**9. LINQ Query Expressions (C# Programming Guide)**

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language (also in Visual Basic and potentially any other .NET language). With LINQ, a query is now a first-class language construct, just like classes, methods, events and so on.

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax* introduced in C# 3.0. By using query syntax, you can perform even complex filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO.NET Datasets, XML documents and streams, and .NET collections.

**10. Partial Methods**

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods enable the implementer of one part of a class to define a method, similar to an event. The implementer of the other part of the class can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

* Partial method declarations must begin with the contextual keyword [partial](http://msdn.microsoft.com/en-us/library/wbx7zzdd.aspx) and the method must return [void](http://msdn.microsoft.com/en-us/library/yah0tteb.aspx).
* Partial methods can have [ref](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) but not [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameters.
* Partial methods are implicitly [private](http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx), and therefore they cannot be [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4.aspx).
* Partial methods cannot be [extern](http://msdn.microsoft.com/en-us/library/e59b22c5.aspx), because the presence of the body determines whether they are defining or implementing.
* Partial methods can have [static](http://msdn.microsoft.com/en-us/library/98f28cdx.aspx) and [unsafe](http://msdn.microsoft.com/en-us/library/chfa2zb8.aspx) modifiers.
* Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
* You can make a [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e.aspx) to a partial method that has been defined and implemented, but not to a partial method that has only been defined.