C# class

Fields

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

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

Fields allow the following modifiers:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifier	`new`
Unsafe code modifier	`unsafe`
Read-only modifier	`readonly`
Threading modifier	`volatile`

The readonly modifier

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

Field initialization

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

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

Declaring multiple fields together

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

```
static readonly int legs = 8,

                    eyes = 2;
```

Methods

A method performs an action in a series of statements. A method can receive *input* data from the caller by
specifying *parameters* and *output* data back to the caller by specifying a *return type*. A method can specify a `void` return type, indicating that it doesn't return any value to its caller. A method can also output data back to the caller via `ref`/`out` parameters.
A method's *signature* must be unique within the type. A method's signature comprises its name and parameter types in order (but not the parameter *names*, nor the return type).

Methods allow the following modifiers:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifiers	`new virtual abstract override sealed`
Partial method modifier	`partial`
Unmanaged code modifiers	`unsafe extern`
Asynchronous code modifier	`async`

Expression-bodied methods (C# 6)

A method that comprises a single expression, such as the following:

```
int Foo (int x) { return x * 2; }
```

can be written more tersely as an *expression-bodied method*. A fat arrow replaces the braces and `return` keyword:

```
  int Foo (int x) => x * 2;
```

Expression-bodied functions can also have a void return type:

```
  void Foo (int x) => Console.WriteLine (x);
```

Overloading methods

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

```
  void Foo (int x) {...}
  void Foo (double x) {...}
  void Foo (int x, float y) {...}
  void Foo (float x, int y) {...}
```

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

```
  void  Foo (int x) {...}
  float Foo (int x) {...}           // Compile-time error

  void  Goo (int[] x) {...}
  void  Goo (params int[] x) {...}  // Compile-time error
```

Pass-by-value versus pass-by-reference

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

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

Local methods (C# 7)

From C# 7, you can define a method inside another method:

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  Console.WriteLine (Cube (4));
  Console.WriteLine (Cube (5));

  int Cube (int value) => value * value * value;
}
```

The local method (`Cube`, in this case) is visible only to the enclosing method (`WriteCubes`). This simplifies the containing type and instantly signals to anyone looking at the code that `Cube` is used nowhere else. Another benefit of local methods is that they can access the local variables and parameters of the enclosing method.

Instance Constructors

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

```
public class Panda
{
  string name;                   // Define field
  public Panda (string n)        // Define constructor
  {
    name = n;                    // Initialization code
(set up field)
  }
}
...

Panda p = new Panda ("Petey");   // Call constructor
```

Instance constructors allow the following modifiers:

Access modifiers	`public internal private protected`
Unmanaged code modifiers	`unsafe extern`

From C# 7, single-statement constructors can also be written as expression-bodied members:

```
public Panda (string n) => name = n;
```

Overloading constructors

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

```
using System;

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

When one constructor calls another, the *called constructor* executes first. You can pass an *expression* into another constructor as follows:

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

The expression itself cannot make use of the `this` reference, for example, to call an instance method. (This is enforced because the object has not been initialized by the constructor at this stage, so any methods that you call on it are likely to fail.) It can, however, call static methods.

Implicit parameterless constructors

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

Constructor and field initialization order

We saw previously that fields can be initialized with default values in their declaration:

```
class Player
{
  int shields = 50;   // Initialized first
  int health = 100;   // Initialized second
}
```

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

Deconstructors (C# 7)

C# 7 introduces the *deconstructor* pattern. A deconstructor (also called a *deconstructing method*) acts as an approximate opposite to a constructor: Whereas a constructor typically takes a set of values (as parameters) and assigns them to fields, a deconstructor does the reverse and assigns fields back to a set of variables.
A deconstruction method must be called `Deconstruct`, and have one or more `out` parameters, such as in the following class:

```
class Rectangle
{
  public readonly float Width, Height;

  public Rectangle (float width, float height)
  {
    Width = width;
    Height = height;
  }

  public void Deconstruct (out float width, out float height)
  {
    width = Width;
    height = Height;
  }
}
```

To call the deconstructor, we use the following special syntax:

```
var rect = new Rectangle (3, 4);

(float width, float height) = rect;          //
Deconstruction
Console.WriteLine (width + " " + height);    // 3 4
```

The second line is the deconstructing call. It creates two local variables and then calls the `Deconstruct` method. Our deconstructing call is equivalent to:

```
float width, height;
rect.Deconstruct (out width, out height);
```

Or:

```
rect.Deconstruct (out var width, out var height);
```

Deconstructing calls allow implicit typing, so we could shorten our call to:

```
(var width, var height) = rect;
```

Or simply:

```
var (width, height) = rect;
```

If the variables into which you're deconstructing are already defined, omit the types altogether:

```
float width, height;

(width, height) = rect;
```

This is called a *deconstructing assignment*.
You can offer the caller a range of deconstruction options by overloading the `Deconstruct` method.

Object Initializers

To simplify object initialization, any accessible fields or properties of an object can be set via an *object initializer* directly after construction. For example, consider the following class:

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

```
    public bool LikesHumans;

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

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

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

Properties

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

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

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

```
  public class Stock
  {
    decimal currentPrice;           // The private
  "backing" field

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

`get` and `set` denote property *accessors*. The `get` accessor runs when the property is read. It must return a value of the property's
type. The `set` accessor runs when the property is assigned. It has an

implicit parameter named `value` of the property's type that you typically assign to a private field (in this case, `currentPrice`).
Although properties are accessed in the same way as fields, they differ in that they give the implementer complete control over getting and setting its value. This control enables the implementer to choose whatever internal representation is needed, without exposing the internal details to the user of the property. In this example, the `set` method could throw an exception if `value` was outside a valid range of values.
Properties allow the following modifiers:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifiers	`new virtual abstract override sealed`
Unmanaged code modifiers	`unsafe extern`

Read-only and calculated properties

A property is read-only if it specifies only a `get` accessor, and it is write-only if it specifies only a `set` accessor. Write-only properties are rarely used.
A property typically has a dedicated backing field to store the underlying data. However, a property can also be computed from other data. For example:

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
  get { return currentPrice * sharesOwned; }
}
```

Expression-bodied properties (C# 6, C# 7)

From C# 6, you can declare a read-only property, such as the preceding example, more tersely as an *expression-bodied property*. A fat arrow replaces all the braces and the `get` and `return` keywords:

```
public decimal Worth => currentPrice * sharesOwned;
```

C# 7 extends this further by allowing `set` accessors to be expression-bodied, with a little extra syntax:

```
public decimal Worth
{
  get => currentPrice * sharesOwned;
  set => sharesOwned = value / currentPrice;
}
```

Automatic properties

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

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

The compiler automatically generates a private backing field of a compiler-generated name that cannot be referred to. The `set` accessor can be marked `private` or `protected` if you want to expose the property as read-only to other types.

Property initializers (C# 6)

From C# 6, you can add a *property initializer* to automatic properties, just as with fields:

```
public decimal CurrentPrice { get; set; } = 123;
```

This gives `CurrentPrice` an initial value of 123. Properties with an initializer can be read-only:

```
public int Maximum { get; } = 999;
```

Just as with read-only fields, read-only automatic properties can also be assigned in the type's constructor. This is useful in creating *immutable* (read-only) types.

get and set accessibility

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

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

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

CLR property implementation

C# property accessors internally compile to methods called `get_`*`XXX`* and `set_`*`XXX`*:

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

Simple nonvirtual property accessors are *inlined* by the JIT (Just-In-Time) compiler, eliminating any performance difference between accessing a property and a field. Inlining is an optimization in which a method call is replaced with the body of that method.
With WinRT properties, the compiler assumes the `put_`*`XXX`* naming convention rather than `set_`*`XXX`*.

Indexers

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

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

The syntax for using indexers is like that for using arrays, except that the index argument(s) can be of any type(s).

Implementing an indexer

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

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

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

Here’s how we could use this indexer:

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

If you omit the `set` accessor, an indexer becomes read-only, and expression-bodied syntax may be used in C# 6 to shorten its definition:

```
  public string this [int wordNum] => words [wordNum];
```

Using a string as an indexer value

```
class DayCollection
{
   string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

   // This method finds the day or returns -1
   private int GetDay(string testDay)
   {

      for (int j = 0; j < days.Length; j++)
      {
         if (days[j] == testDay)
         {
            return j;
         }
      }

      throw new System.ArgumentOutOfRangeException(testDay, "testDay must
be in the form \"Sun\", \"Mon\", etc");
   }

   // The get accessor returns an integer for a given string
   public int this[string day]
   {
      get
      {
         return (GetDay(day));
      }
   }
}

class Program
{
   static void Main(string[] args)
   {
      DayCollection week = new DayCollection();
      System.Console.WriteLine(week["Fri"]);

      // Raises ArgumentOutOfRangeException
      System.Console.WriteLine(week["Made-up Day"]);

      // Keep the console window open in debug mode.
```

```
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
// Output: 5
```

Static Constructors

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

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

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

- Instantiating the type
- Accessing a static member in the type

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

Static Classes

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

Finalizers

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

```
class Class1
{
  ~Class1()
```

```
    {
      ...
    }
  }
```

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

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

Partial Types and Methods

Partial types allow a type definition to be split—typically across multiple files. A common scenario is for a partial class to be auto-generated from some other source (such as a Visual Studio template or designer), and for that class to be augmented with additional hand-authored methods. For example:

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

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

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

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

Participants cannot have conflicting members. A constructor with the same parameters, for instance, cannot be repeated. Partial types are resolved entirely by the compiler, which means that each participant must be available at compile time and must reside in the same assembly. You can specify a base class on one or more partial class declarations, as long as the base class, if specified, is the same. In addition, each participant can independently specify interfaces to implement. We cover base classes and interfaces in "Inheritance" and "Interfaces".

The compiler makes no guarantees with regard to field initialization order between partial type declarations.

Partial methods

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

```
partial class PaymentForm    // In auto-generated file
{
  ...
  partial void ValidatePayment (decimal amount);
}

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

}
```

A partial method consists of two parts: a *definition* and an *implementation*. The definition is typically written by a code generator, and the implementation is typically manually authored. If an implementation is not provided, the definition of the partial method is compiled away (as is the code that calls it). This allows auto-generated code to be liberal in providing hooks, without having to worry about bloat. Partial methods must be `void` and are implicitly `private`. Partial methods were introduced in C# 3.0.