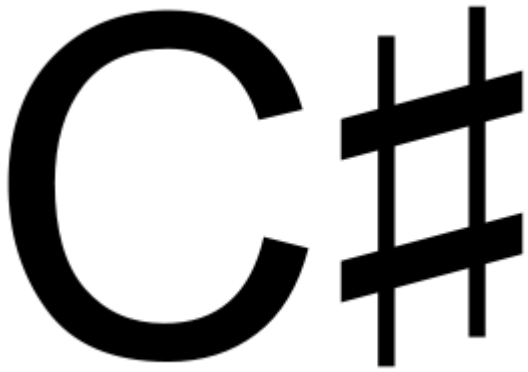
C# Programming

by Wikibooks contributors



Developed on **Wikibooks**,
the open-content textbooks collection

Principal authors: Rod A. Smith (C) · Jonas Nordlund (C) · Jlenthe (C) · Nercury (C) · Ripper234 (C)

Cover: C♯ musical note, by Mothmolevna (*See naming*) (GFDL)

The current version of this Wikibook may be found at:
http://en.wikibooks.org/wiki/C_Sharp_Programming

Contents

1 Foreword

live version · discussion · edit chapter · comment · report an error

C# (pronounced "See Sharp") is a multi-purpose computer programming language suitable for all development needs.

Introduction

Although C# is derived from the C programming language, it has features such as garbage collection that allow beginners to become proficient in C# more quickly than in C or C++. Similar to Java, it is object-oriented, comes with an extensive *class library*, and supports exception handling, multiple types of polymorphism, and separation of interfaces from implementations. Those features, combined with its powerful development tools, multi-platform support, and *generics*, make C# a good choice for many types of software development projects: rapid application development projects, projects implemented by individuals or large or small teams, Internet applications, and projects with strict reliability requirements. Testing frameworks such as NUnit make C# amenable to test-driven development and thus a good language for use with Extreme Programming (XP). Its strong typing helps to prevent many programming errors that are common in weakly typed languages.

A large part of the power of C# (as with other .NET languages), comes with the common .NET Framework API, which provides a large set of classes, including ones for encryption, TCP/IP socket programming, and graphics. Developers can thus write part of an application in C# and another part in another .NET language (e.g. VB .NET), keeping the tools, library, and object-oriented development model while only having to learn the new language syntax.

Because of the similarities between C# and the C family of languages, as well as Java, a developer with a background in object-oriented languages like C++ may find C# structure and syntax intuitive.

Standard

Microsoft, Anders Hejlsberg as Chief Engineer, created C# as part of their .NET initiative and subsequently opened its specification via the ECMA. Thus, the language is open to implementation by other parties. Other implementations include Mono and DotGNU.

C# and other .NET languages rely on an implementation of the virtual machine specified in the Common Language Infrastructure, like Microsoft's *Common Language Runtime* (CLR). That virtual machine manages memory, handles object references, and performs Just-In-Time (JIT) compiling of Common Intermediate Language code. The virtual machine makes C# programs safer

than those that must manage their own memory and is one of the reasons .NET language code is referred to as *managed code*. More like Java than C and C++, C# discourages explicit use of pointers, which could otherwise allow software bugs to corrupt system memory and force the operating system to halt the program forcibly with nondescript error messages.

History

Microsoft's original plan was to create a rival to Java, named J++ but this was abandoned to create C#, codenamed "Cool".

Microsoft submitted C# to the ECMA standards group mid-2000.

C# 2.0 was released in late-2005 as part of Microsoft's development suite, Visual Studio 2005. The 2.0 version of C# includes such new features as generics, partial classes, and iterators.

Se microsoft-watch and hitmil.

2 Getting Started

live version · discussion · edit chapter · comment · report an error

To compile your first C# application, you will need a copy of a .NET Framework SDK installed on your PC.

There are two .NET frameworks available: Microsoft's and Mono's.

Microsoft
: For Windows, the .Net Framework SDK can be downloaded from Microsoft's .NET Framework Developer Center. If the default Windows directory (the directory where Windows or WinNT is installed) is `C:\WINDOWS`, the .Net Framework SDK installation places the Visual C# .NET Compiler (`csc`) in the `C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705` directory for version 1.0, the `C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322` directory for version 1.1, **or** the `C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727` directory for version 2.0.

Mono
: For Windows, Linux, or other Operating Systems, an installer can be downloaded from the Mono website.
For Linux, a good compiler is cscc which can be downloaded for free from the DotGNU Portable.Net project page. The compiled programs can then be run with ilrun.

If you are working on Windows it is a good idea to add the path to the folders that contain cs.exe or mcs.exe to the Path environment variable so that you do not need to type the full path each time you want to compile.

For writing C#.NET code, there are plenty of editors that are available. It's entirely possible to write C#.NET programs with a simple text editor, but it should be noted that this requires you to compile the code yourself. Microsoft offers a wide range of code editing programs under the Visual Studio line that offer syntax highlighting as well as compiling and debugging capabilities. Currently C#.NET can be compiled in Visual Studio 2002 and 2003 (only supports the .NET Framework version 1.0 and 1.1) and Visual Studio 2005 (supports the .NET Framework 2.0 and earlier versions with some tweaking). Microsoft offers five Visual Studio editions, four of which cost money. The Visual Studio C# Express Edition can be downloaded and used for free from Microsoft's website.

The code below will demonstrate a C# program written in a simple text editor. Start by saving the following code to a text file called `hello.cs`:

```
using System;

namespace MyConsoleApplication
{
        class MyFirstClass
```

```
        {
                static void Main(string[] args)
                {
                        System.Console.WriteLine("Hello,");
                        Console.WriteLine("World!");
                        Console.ReadLine();
                }
        }
}
```

To compile `hello.cs`, run the following from the command line:

- For standard Microsoft installations of .Net 2.0, run `C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\csc.exe hello.cs`
- For Mono run `mcs hello.cs`.
- For users of cscc, compile with "cscc -o <name>.exe <name>.cs".

Doing so will produce `hello.exe`. The following command will run `hello.exe`:

- On Windows, use `hello.exe`.
- On Linux, use `mono hello.exe` or "ilrun <name>.exe".

Alternatively, in Visual C# express, you could just hit F5 or the green play button to run the code, even though that is for debugging.

Running `hello.exe` will produce the following output:

```
Hello,
World!
```

The program will then wait for you to strike 'enter' before returning to the command prompt.

Note that the example above includes the `System` namespace via the `using` keyword. That inclusion allows direct references to any member of the `System` namespace without specifying its fully qualified name.

The first call to the `WriteLine` method of the `Console` class uses a fully qualified reference.

```
System.Console.WriteLine("Hello,");
```

The second call to that method shortens the reference to the `Console` class by taking advantage of the fact that the `System` namespace is included (with `using System`).

```
Console.WriteLine("World!");
```

C# is a fully object-oriented language. The following sections explain the syntax of the C# language as a beginner's course for programming in the language. Note that much of the power of the language comes from the classes provided with the .Net framework, which are not part of the C# language syntax

per se.

live version · discussion · edit chapter · comment · report an error

3 Syntax

live version · discussion · edit chapter · comment · report an error

C# syntax looks quite similar to the syntax of Java because both inherit much of their syntax from C and C++. **The object-oriented** nature of C# requires the high-level structure of a C# program to be defined in terms of classes, whose detailed behaviors are defined by their *statements*.

Statements

The basic unit of execution in a C# program is the *statement*. A statement can declare a variable, define an expression, perform a simple action by calling a method, control the flow of execution of other statements, create an object, or assign a value to a variable, property, or field. Statements are usually terminated by a semicolon.

Statements can be grouped into comma-separated statement lists or brace-enclosed statement blocks.

Examples:

```
int sampleVariable;                          // declaring a variable
sampleVariable = 5;                          // assigning a value
SampleClass sampleObject = new SampleClass(); // constructing a new object
sampleObject.SampleInstanceMethod();         // calling an instance method
SampleClass.SampleStaticMethod();            // calling a static method

// executing a "for" loop with an embedded "if" statement
for(int i = 0; i < upperLimit; i++)
{
    if (SampleClass.SampleStaticMethodReturningBoolean(i))
    {
        sum += sampleObject.SampleMethodReturningInteger(i);
    }
}
```

Statement blocks

A series of statements surrounded by curly braces form a *block* of code. Among other purposes, code blocks serve to limit the scope of variables defined within them. Code blocks can be nested and often appear as the bodies of methods, the protected statements of a `try` block, and the code within a corresponding `catch` block.

```
private void MyMethod()
{
    // This block of code is the body of "MyMethod()"
    CallSomeOtherMethod();

    try
    {
        // Here is a code block protected by the "try" statement.
```

```
            int variableWithLimitedScope;

            // "variableWithLimitedScope" is accessible in this code block.
        }
        catch(Exception)
        {
            // Here is yet another code block.

            // "variableWithLimitedScope" is not accessible here.
        }

        // "variableWithLimitedScope" is not accessible here, either.

        CallYetAnotherMethod();

        // Here ends the code block for the body of "MyMethod()".
    }
```

Comments

Comments allow inline documentation of source code. The C# compiler ignores comments. Three styles of comments are allowed in C#:

Single-line comments
: The "//" character sequence marks the following text as a single-line comment. Single-line comments, as one would expect, end at the first end-of-line following the "//" comment marker.

Multiple-line comments
: Comments can span multiple lines by using the multiple-line comment style. Such comments start with "/*" and end with "*/". The text between those multi-line comment markers is the comment.

```
//This style of a comment is restricted to one line.
/*
   This is another style of a comment.
   It allows multiple lines.
*/
```

XML Documentation-line comments
: This comment is used to generate XML documentation. Each line of the comment begins with "///".

```
/// <summary> documentation here </summary>
```

This is the most recommended type. Avoid using butterfly style comments. For example:

```
//*************************
// Butterfly style documentation comments like this are not recommended.
//*************************
```

Case sensitivity

C# is case-sensitive, including its variable and method names.

The variables `myInteger` and `MyInteger` below are distinct because C# is case-sensitive:

```
int myInteger = 3;
int MyInteger = 5;
```

The following code will generate a compiler error (unless a custom class or variable named `console` has a method named `writeline()`):

```
// Compiler error!
console.writeline("Hello");
```

The following corrected code compiles as expected because it uses the correct case:

```
Console.WriteLine("Hello");
```

4 Variables

live version · discussion · edit chapter · comment · report an error

Variables are used to store values. More technically, a variable binds an object (in the general sense of the term, i.e. a specific value) to an identifier (the variable's name) so that the object can be accessed later. Variables can, for example, store the value of user input:

```
string name = Console.ReadLine();
Console.WriteLine ( "Good morning, {0}" , name );
```

Each variable is declared with an explicit *type*. Only values whose types are compatible with the variable's declared type can be bound to (stored in) the variable.

Fields, Local Variables, and Parameters

C# supports several program elements corresponding to the general programming concept of *variable*: *fields*, *parameters*, and *local variables*.

Fields

Fields, sometimes called class-level variables, are variables associated with classes or structures. An *instance variable* is a field associated with an instance of the class or structure, while a *static variable*, declared with the **`static`** keyword, is a field associated with the type itself. Fields can also be associated with their class by making them *constants* (**`const`**), which requires a declaration assignment of a constant value and prevents subsequent changes to the field.

Each field has a visibility of *public*, *protected*, *internal*, *protected internal*, or *private* (from most visible to least visible).

Local variables

Like fields, local variables can optionally be *constant* (**`const`**). Constant local variables are stored in the assembly data region, while non-constant local variables are stored (or referenced from) the stack. They thus have both a scope and an extent of the method or statement block that declares them.

Parameter

Parameters are variables associated with a method.

An *in* parameter may either have its value passed in from the callee to the method's environment, so that changes to the parameter by the method do not affect the value of the callee's variable, or passed in by reference, so that changes to the variables will affect the value of the callee's variable. Value types (int, double, string) are passed in "by value" while reference types (objects) are passed in "by reference."

An *out* parameter does not have its value copied, thus changes to the variable's value within the method's environment directly affect the value from the callee's environment. Such a variable is considered by the compiler to be *unbound* upon method entry, thus it is illegal to reference an *out* parameter before assigning it a value. It also **must** be assigned by the method in each valid (non-exceptional) code path through the method in order for the method to compile.

A *reference* parameter is similar to an *out* parameter, except that it is *bound* before the method call and it need not be assigned by the method.

A *params* parameter represents a variable number of parameters. If a method signature includes one, the *params* argument must be the last argument in the signature.

Types

Each **type** in C# is either a *value type* or a *reference type*. C# has several predefined ("built-in") types and allows for declaration of custom value types and reference types.

Integral types

Because the type system in C# is unified with other languages that are CLI-compliant, each integral C# type is actually an alias for a corresponding type in the .NET framework. Although the names of the aliases vary between .NET languages, the underlying types in the .NET framework remain the same. Thus, objects created in assemblies written in other languages of the .NET Framework can be bound to C# variables of any type to which the value can be converted, per the conversion rules below. The following illustrates the cross-language compatibility of types by comparing C# code with the equivalent Visual Basic .NET code:

```
// C#
public void UsingCSharpTypeAlias()
{
  int i = 42;
}
public void EquivalentCodeWithoutAlias()
{
  System.Int32 i = 42;
}
```

```
' Visual Basic .NET
Public Sub UsingVisualBasicTypeAlias()
  Dim i As Integer = 42
End Sub
Public Sub EquivalentCodeWithoutAlias()
  Dim i As System.Int32 = 42
End Sub
```

Using the language-specific type aliases is often considered more readable than using the fully-qualified .NET Framework type names.

The fact that each C# type corresponds to a type in the unified type system gives each *value type* a consistent size across platforms and compilers. That consistency is an important distinction from other languages such as C, where, e.g. a `long` is only guaranteed to be at *least as large as an* `int`, and is implemented with different sizes by different compilers. As *reference* types, variables of types derived from `object` (i.e. any `class`) are exempt from the consistent size requirement. That is, the size of *reference* types like `System.IntPtr`, as opposed to *value* types like `System.Int`, may vary by platform. Fortunately, there is rarely a need to know the actual size of a *reference* type.

There are two predefined *reference types*: `object`, an alias for the `System.Object` class, from which all other reference types derive; and `string`, an alias for the `System.String` class. C# likewise has several integral value types, each an alias to a corresponding value type in the `System` namespace of the .NET Framework. The predefined C# type aliases expose the methods of the underlying .NET Framework types. For example, since the .NET Framework's `System.Int32` type implements a `ToString()` method to convert the value of an integer to its string representation, C#'s `int` type exposes that method:

```
int i = 97;
string s = i.ToString();
// The value of s is now the string "97".
```

Likewise, the `System.Int32` type implements the `Parse()` method, which can therefore be accessed via C#'s `int` type:

```
string s = "97";
int i = int.Parse(s);
// The value of i is now the integer 97.
```

The unified type system is enhanced by the ability to convert value types to reference types (*boxing*) and likewise to convert certain reference types to their corresponding value types (*unboxing*):

```
object boxedInteger = 97;
int unboxedInteger = (int)boxedInteger;
```

The built-in C# type aliases and their equivalent .NET Framework types follow:

Integers

C# Alias	.NET Type	Size (bits)	Range
sbyte	System.SByte	8	-128 to 127
byte	System.Byte	8	0 to 255
short	System.Int16	16	-32,768 to 32,767
ushort	System.UInt16	16	0 to 65,535
char	System.Char	16	A unicode character of code 0 to 65,535
int	System.Int32	32	-2,147,483,648 to 2,147,483,647
uint	System.UInt32	32	0 to 4,294,967,295
long	System.Int64	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	System.UInt64	64	0 to 18,446,744,073,709,551,615

Floating-point

C# Alias	.NET Type	Size (bits)	Precision	Range
float	System.Single	32	7 digits	1.5×10^{-45} to 3.4×10^{38}
double	System.Double	64	15-16 digits	5.0×10^{-324} to 1.7×10^{308}
decimal	System.Decimal	128	28-29 decimal places	1.0×10^{-28} to 7.9×10^{28}

Other predefined types

C# Alias	.NET Type	Size (bits)	Range
bool	System.Boolean	32	true or false, which aren't related to any integer in C#.
object	System.Object	32/64	Platform dependant (a pointer to an object).
string	System.String	16 * length	A unicode string with no special upper bound.

Custom types

The predefined types can be aggregated and extended into custom types.

Custom *value types* are declared with the **`struct`** or **`enum`** keyword. Likewise, custom reference types are declared with the **`class`** keyword.

Arrays

Although the number of dimensions is included in array declarations, the size of each dimension is not:

```
string[] s ;
```

Assignments to an array variable (prior to the variable's usage), however, specify the size of each dimension:

```
s = new string[5] ;
```

As with other variable types, the declaration and the initialization can be combined:

```
string[] s = new string[5] ;
```

It is also important to note that like in Java, arrays are passed by reference, and not passed by value. For example, the following code snippet successfully swaps two elements in an integer array:

```
static void swap (int[] arr, int i, int j)
{
          int temp = arr[i];
          arr[i] = arr[j];
          arr[j] = temp;
}
```

Conversion

Values of a given type may or may not be explicitly or implicitly convertible to other types depending on predefined conversion rules, inheritance structure, and explicit cast definitions.

Predefined conversions

Many predefined value types have predefined conversions to other predefined value types. If the type conversion is guaranteed not to lose information, the conversion can be *implicit* (i.e. an explicit *cast* is not required).

Inheritance polymorphism

A value can be implicitly converted to any class from which it inherits or interface that it implements. To convert a base class to a class that inherits from it, the conversion must be explicit in order for the conversion statement to compile. Similarly, to convert an interface instance to a class that implements it, the conversion must be explicit in order for the conversion statement to compile. In either case, the runtime environment throws a conversion exception if the value to convert is not an instance of the target type or any of its derived types.

Scope and extent

The scope and extent of variables is based on their declaration. The scope of parameters and local variables corresponds to the declaring method or statement block, while the scope of fields is associated with the instance or class and is potentially further restricted by the field's access modifiers.

The extent of variables is determined by the runtime environment using implicit reference counting and a complex garbage collection algorithm.

5 Operators

live version · discussion · edit chapter · comment · report an error

C# operators and their precedence closely resemble the operators in other languages of the C family.

Similar to C++, classes can *overload* most operators, defining or redefining the behavior of the operators in contexts where the first argument of that operator is an instance of that class, but doing so is often discouraged for clarity.

Following are the built-in behaviors of C# operators.

Arithmetic

The following arithmetic operators operate on numeric operands (arguments `a` and `b` in the "sample usage" below).

Sample usage	Read	Explanation
`a + b`	`a` *plus* `b`	The binary operator `+` returns the sum of its arguments.
`a - b`	`a` *minus* `b`	The binary operator `-` returns the difference between its arguments.
`a * b`	`a` *times* `b`	The binary operator `*` returns the multiplicative product of its arguments.
`a / b`	`a` *divided by* `b`	The binary operator `/` returns the quotient of its arguments. If both of its operators are integers, it obtains that quotient using *integer division* (i.e. it drops any resulting remainder).
`a % b`	`a` *mod* `b`	The binary operator `%` operates only on integer arguments. It returns the remainder of *integer division* of those arguments. *(See modular arithmetic.)*
`a++`	`a` *plus plus*	The unary operator `++` operates only on arguments that have an *l-value*. When placed **after** its argument, it increments that argument by 1 and returns the value of that argument before it was incremented.
`++a`	*plus plus* `a`	The unary operator `++` operates only on arguments that have an *l-value*. When placed **before** its argument, it increments that argument by 1 and returns the resulting value.
`a--`	`a` *minus minus*	The unary operator `--` operates only on arguments that have an *l-value*. When placed **after** its argument, it decrements that argument by 1 and returns the value of

		that argument before it was decremented.
`--a`	*minus minus* `a`	The unary operator `--` operates only on arguments that have an *l-value*. When placed **before** its argument, it decrements that argument by 1 and returns the resulting value.

Logical

The following logical operators operate on boolean or integral operands, as noted.

Sample usage	Read	Explanation
`a & b`	`a` *bitwise and* `b`	The binary operator & evaluates both of its operands and returns the logical conjunction ("AND") of their results. If the operands are integral, the logical conjunction is performed bitwise.
`a && b`	`a` *and* `b`	The binary operator && operates on boolean operands only. It evaluates its first operand. If the result is *false*, it returns *false*. Otherwise, it evaluates and returns the results of the second operand. Note that if evaluating the second operand would hypothetically have no side effects, the results are identical to the logical conjunction performed by the & operator.
`a | b`	`a` *bitwise or* `b`	The binary operator | evaluates both of its operands and returns the logical disjunction ("OR") of their results. If the operands are integral, the logical disjunction is performed bitwise.
`a || b`	`a` *or* `b`	The binary operator || operates on boolean operands only. It evaluates the first operand. If the result is *true*, it returns *true*. Otherwise, it evaluates and returns the results of the second operand. Note that if evaluating the second operand would hypothetically have no side effects, the results are identical to the logical disjunction performed by the | operator.
`a ^ b`	`a` *x-or* `b`	The binary operator ^ returns the exclusive or ("XOR") of their results. If the operands are integral, the exclusive or is performed bitwise.
`!a`	*not* `a`	The unary operator ! operates on a boolean operand only. It evaluates its operand and returns the negation ("NOT") of the result. That is, it returns *true* if `a` evaluates to *false* and it returns *false* if `a` evaluates to *true*.
`~a`	*bitwise*	The unary operator ~ operates on integral operands only. It

	not a	evaluates its operand and returns the bitwise negation of the result. That is, ~a returns a value where each bit is the negation of the corresponding bit in the result of evaluating a.

Bitwise shifting

Sample usage	Read	Explanation
a << b	a *left shift* b	The binary operator << evaluates its operands and returns the resulting first argument left-shifted by the number of bits specified by the second argument. It discards high-order bits that shift beyond the size of its first argument and sets new low-order bits to zero.
a >> b	a *right shift* b	The binary operator >> evaluates its operands and returns the resulting first argument right-shifted by the number of bits specified by the second argument. It discards low-order bits that are shifted beyond the size of its first argument and sets new high-order bits to the sign bit of the first argument, or to zero if the first argument is unsigned.

Relational

The binary relational operators ==, !=, <, >, <=, and >= are used for relational operations and for type comparisons.

Sample usage	Read	Explanation
a == b	a *is equal to* b	For arguments of *value* type, the operator == returns *true* if its operands have the same value, false otherwise. For the *string* type, it returns *true* if the strings' character sequences match. For other *reference* types (types derived from System.Object), however, a == b returns *true* only if a and b reference the same object.
a != b	a *is not equal to* b	The operator != returns the logical negation of the operator ==. Thus, it returns *true* if a is not equal to b, and *false* if they are equal.
a < b	a *is less than* b	The operator < operates on integral types. It returns *true* if a is less than b, *false* otherwise.
a > b	a *is greater than* b	The operator > operates on integral types. It returns *true* if a is greater than b, *false* otherwise.
a <= b	a *is less*	The operator <= operates on integral types. It returns

	than or equal to b	*true* if a is less than or equal to b, *false* otherwise.
a >= b	a *is greater than or equal to* b	The operator >= operates on integral types. It returns *true* if a is greater than or equal to b, *false* otherwise.

Assignment

The assignment operators are binary. The most basic is the operator =. Not surprisingly, it assigns the value of its second argument to its first argument.

(More technically, the operator = requires for its first (*left*) argument an expression to which a value can be assigned (an *l-value*) and for its second (*right*) argument an expression which can be evaluated (an *r-value*). That requirement of an *assignable* expression to its left and a *bound* expression to its right is the origin of the terms *l-value* and *r-value*.)

The first argument of the assignment operator (=) is typically a variable. When that argument has a *value* type, the assignment operation changes the argument's underlying value. When the first argument is a *reference* type, the assignment operation changes the reference, so the first argument typically just refers to a different object but the object that it originally referenced does not change (except that it may no longer be referenced and may thus be a candidate for *garbage collection*).

Sample usage	Read	Explanation
a = b	a *equals* (or *set to*) b	The operator = evaluates its second argument and then assigns the results to (the *l-value* indicated by) its first argument.
a = b = c	b *set to* c, and then a *set to* b	Equivalent to a = (b = c). When there are consecutive assignments, the right-most assignment is evaluated first, proceeding from right to left. In this example, both variables a and b have the value of c.

Short-hand Assignment

The short-hand assignment operators shortens the common assignment operation of a = a *operator* b into a *operator*= b, resulting in less typing and neater syntax.

Sample usage	Read	Explanation
a += b	a *plus equals* (or *increment by*) b	Equivalent to a = a + b.

`a -= b`	`a` *minus equals* (or *decrement by*) `b`	Equivalent to `a = a - b`.
`a *= b`	`a` *multiply equals* (or *multiplied by*) `b`	Equivalent to `a = a * b`.
`a /= b`	`a` *divide equals* (or *divided by*) `b`	Equivalent to `a = a / b`.
`a %= b`	`a` *mod equals* `b`	Equivalent to `a = a % b`.
`a &= b`	`a` *and equals* `b`	Equivalent to `a = a & b`.
`a |= b`	`a` *or equals* `b`	Equivalent to `a = a | b`.
`a ^= b`	`a` *xor equals* `b`	Equivalent to `a = a ^ b`.
`a <<= b`	`a` *left-shift equals* `b`	Equivalent to `a = a << b`.
`a >>= b`	`a` *right-shift equals* `b`	Equivalent to `a = a >> b`.

Type information

Expression	Explanation
`x is T`	returns true if the variable `x` of base class type stores an object of derived class type T, or, if `x` is of type `T`. Else returns false.
`x as T`	returns `(T)x` *(x casted to T)* if the variable `x` of base class type stores an object of derived class type `T`, or, if `x` is of type `T`. Else returns null. Equivalent to `x is T ? (T)x : null`
sizeof(x)	returns the size of the value type `x`. Remarks: The sizeof operator can be applied only to value types, not reference types. The sizeof operator can only be used in the unsafe mode.
typeof(T)	returns a `System.Type` object describing the type. `T` must be the name of the type, and not a variable. Use the `GetType` method to retrieve run-time type information of variables.

Pointer manipulation

Expression	Explanation
To be done	*, ->, [], &

Overflow exception control

Expression	Explanation
checked(a)	uses overflow checking on value a
unchecked(a)	avoids overflow checking on value a

Others

Expression	Explanation
a.b	accesses member b of type or namespace a
a[b]	the value of index b in a
(a)b	casts the value b to type a
new a	creates an object of type a
a + b	if a and b are string types, concatenates a and b
a ? b : c	if a is true, returns the value of b, otherwise c
a ?? b	if a is null, returns b, otherwise returns a

6 Data Structures

There are various ways of grouping sets of data together in C#.

Enumerations

An *enumeration* is a data type that *enumerates* a set of items by assigning to each of them an identifier (a name), while exposing an underlying base type for ordering the elements of the enumeration. The underlying type is `int` by default, but can be any one of the integral types except for `char`.

Enumerations are declared as follows:

```
enum Weekday { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

The elements in the above enumeration are then available as constants:

```
Weekday day = Weekday.Monday;
if (day == Weekday.Tuesday)
{
    Console.WriteLine("Time sure flies by when you program in C#!");
}
```

If no explicit values are assigned to the enumerated items as the example above, the first element has the value 0, and the successive values are assigned to each subsequent element. However, specific values from the underlying integral type can be assigned to any of the enumerated elements:

```
enum Age { Infant = 0, Teenager = 13, Adult = 18 };

Age age = Age.Teenager;
Console.WriteLine("You become a teenager at an age of {0}.", (int)age);
```

The underlying values of enumerated elements may go unused when the purpose of an enumeration is simply to group a set of items together, e.g., to represent a nation, state, or geographical territory in a more meaningful way than an integer could. Rather than define a group of logically related constants, it is often more readable to use an enumeration.

It may be desirable to create an enumeration with a base type other than `int`. To do so, specify any integral type besides `char` as with base class *extension* syntax after the name of the enumeration, as follows:

```
enum CardSuit : byte { Hearts, Diamonds, Spades, Clubs };
```

Structs

Structures (keyword **struct**) are light-weight objects. They are mostly used when only a data container is required for a collection of value type variables.

Structs are similar to *classes* in that they can have constructors, methods, and even implement interfaces, but there are important differences. *Structs* are value types while *classes* are reference types, which means they behave differently when passed into methods as parameters. Another very important difference is that *structs* cannot support inheritance. While *structs* may appear to be limited with their use, they require less memory and can be less expensive if used in the proper way.

A `struct` can, for example, be declared like this:

```
struct Person
{
    public string name;
    public System.DateTime birthDate;
    public int heightInCm;
    public int weightInKg;
}
```

The `Person` *struct* can then be used like this:

```
Person dana = new Person();
dana.name = "Dana Developer";
dana.birthDate = new DateTime(1974, 7, 18);
dana.heightInCm = 178;
dana.weightInKg = 50;

if (dana.birthDate < DateTime.Now)
{
    Console.WriteLine("Thank goodness! Dana Developer isn't from the future!");
}
```

It is also possible to provide *constructors* to `structs` to make it easier to initialize them:

```
using System;
struct Person
{
    string name;
    DateTime birthDate;
    int heightInCm;
    int weightInKg;

    public Person(string name, DateTime birthDate, int heightInCm, int weightInKg)
    {
        this.name = name;
        this.birthDate = birthDate;
        this.heightInCm = heightInCm;
        this.weightInKg = weightInKg;
    }
}

public class StructWikiBookSample
{
    public static void Main()
    {
        Person dana = new Person("Dana Developer", new DateTime(1974, 7, 18), 178, 50);
    }
}
```

7 Control

Conditional, iteration, jump, and exception handling statements control a program's flow of execution.

An iteration statement can create a loop using keywords such as `do`, `while`, `for`, `foreach`, and `in`.

A jump statement can be used to transfer program control using keywords such as `break`, `continue`, `return`, and `yield`.

An exception handling statement can be used to handle exceptions using keywords such as `throw`, `try-catch`, `try-finally`, and `try-catch-finally`.

Conditional statements

A conditional statement decides whether to execute code based on conditions. The `if` statement and the `switch` statement are the two types of conditional statements in C#.

The `if` statement

As with most of C#, the **`if`** statement has the same syntax as in C, C++, and Java. Thus, it is written in the following form:

if-statement ::= "`if`" "`(`" *condition* "`)`" *if-body* ["`else`" *else-body*]
condition ::= *boolean-expression*
if-body ::= *statement-or-statement-block*
else-body ::= *statement-or-statement-block*

The `if` statement evaluates its *condition* expression to determine whether to execute the *if-body*. Optionally, an `else` clause can immediately follow the *if body*, providing code to execute when the *condition* is *false*. Making the *else-body* another `if` statement creates the common *cascade* of `if`, `else if`, `else if`, `else if`, `else` statements:

```
using System;

public class IfStatementSample
{
    public void IfMyNumberIs()
    {
        int myNumber = 5;
        if ( myNumber == 4 )
            Console.WriteLine("This will not be shown because myNumber is not 4.");
        else if( myNumber < 0 )
        {
            Console.WriteLine("This will not be shown because myNumber is not negative.");
```

```
        }
        else if( myNumber % 2 == 0 )
            Console.WriteLine("This will not be shown because myNumber is not even.");
        else
        {
            Console.WriteLine("myNumber does not match the coded conditions, so this sentence
will be shown!");
        }
    }
}
```

The `switch` statement

The `switch` statement is similar to the statement from C, C++ and Java.

Unlike C, each `case` statement must finish with a jump statement (which can be `break` or `goto` or `return`). In other words, C# does not support "fall through" from one `case` statement to the next (thereby eliminating a common source of unexpected behaviour in C programs). However "stacking" of cases is allowed, as in the example below. If `goto` is used, it may refer to a case label or the default case (e.g. `goto case 0` or `goto default`).

The `default` label is optional. If no default case is defined, then the default behaviour is to do nothing.

A simple example:

```
switch (nCPU)
{
    case 0:
         Console.WriteLine("You don't have a CPU! :-)");
         break;
    case 1:
         Console.WriteLine("Single processor computer");
         break;
    case 2:
         Console.WriteLine("Dual processor computer");
         break;
    // Stacked cases
    case 3:
    case 4:
    case 5:
    case 6:
    case 7:
    case 8:
         Console.WriteLine("A multi processor computer");
         break;
    default:
         Console.WriteLine("A seriously parallel computer");
         break;
}
```

A nice improvement over the C switch statement is that the switch variable can be a string. For example:

```
switch (aircraft_ident)
{
    case "C-FESO":
         Console.WriteLine("Rans S6S Coyote");
         break;
    case "C-GJIS":
         Console.WriteLine("Rans S12XL Airaile");
```

```
            break;
        default:
            Console.WriteLine("Unknown aircraft");
            break;
    }
```

Iteration statements

An iteration statement creates a *loop* of code to execute a variable number of times. The `for` loop, the `do` loop, the `while` loop, and the `foreach` loop are the iteration statements in C#.

The `do...while` loop

The **`do...while`** loop likewise has the same syntax as in other languages derived from C. It is written in the following form:

do...while-loop ::= "`do`" *body* "`while`" "(" *condition* ")"
condition ::= *boolean-expression*
body ::= *statement-or-statement-block*

The `do...while` loop always runs its *body* once. After its first run, it evaluates its *condition* to determine whether to run its *body* again. If the *condition* is *true*, the *body* executes. If the *condition* evaluates to *true* again after the *body* has ran, the *body* executes again. When the *condition* evaluates to *false*, the `do...while` loop ends.

```
using System;

public class DoWhileLoopSample
{
    public void PrintValuesFromZeroToTen()
    {
        int number = 0;
        do
        {
            Console.WriteLine(number++.ToString());
        } while(number <= 10);
    }
}
```

The above code writes the integers from 0 to 10 to the console.

The `for` loop

The **`for`** loop likewise has the same syntax as in other languages derived from C. It is written in the following form:

for-loop ::= "`for`" "(" *initialization* "`;`" *condition* "`;`" *iteration* ")" *body*
initialization ::= *variable-declaration* | *list-of-statements*
condition ::= *boolean-expression*
iteration ::= *list-of-statements*

body ::= *statement-or-statement-block*

The *initialization* variable declaration or statements are executed the first time through the `for` loop, typically to declare and initialize an index variable. The *condition* expression is evaluated before each pass through the *body* to determine whether to execute the body. It is often used to test an index variable against some limit. If the *condition* evaluates to *true*, the *body* is executed. The *iteration* statements are executed after each pass through the *body*, typically to increment or decrement an index variable.

```
public class ForLoopSample
{
    public void ForFirst100NaturalNumbers()
    {
        for(int i=0; i<100; i++)
        {
            System.Console.WriteLine(i.ToString());
        }
    }
}
```

The above code writes the integers from 0 to 99 to the console.

The `foreach` loop

The **`foreach`** statement is similar to the `for` statement in that both allow code to iterate over the items of collections, but the `foreach` statement lacks an iteration index, so it works even with collections that lack indices altogether. It is written in the following form:

foreach-loop ::= "`foreach`" "`(`" *variable-declaration* "`in`" *enumerable-expression* "`)`" *body*
body ::= *statement-or-statement-block*

The *enumerable-expression* is an expression of a type that implements **`IEnumerable`**, so it can be an array or a *collection*. The *variable-declaration* declares a variable that will be set to the successive elements of the *enumerable-expression* for each pass through the *body*. The `foreach` loop exits when there are no more elements of the *enumerable-expression* to assign to the variable of the *variable-declaration*.

```
public class ForEachSample
{
    public void DoSomethingForEachItem()
    {
        string[] itemsToWrite = {"Alpha", "Bravo", "Charlie"};
        foreach (string item in itemsToWrite)
            System.Console.WriteLine(item);
    }
}
```

In the above code, the `foreach` statement iterates over the elements of the string array to write "Alpha", "Bravo", and "Charlie" to the console.

The `while` loop

The **`while`** loop has the same syntax as in other languages derived from C. It is written in the following form:

while-loop ::= "`while`" "`(`" *condition* "`)`" *body*
condition ::= *boolean-expression*
body ::= *statement-or-statement-block*

The `while` loop evaluates its *condition* to determine whether to run its *body*. If the *condition* is *true*, the *body* executes. If the *condition* then evaluates to *true* again, the *body* executes again. When the *condition* evaluates to *false*, the `while` loop ends.

```
using System;

public class WhileLoopSample
{
    public void RunForAwhile()
    {
        TimeSpan durationToRun = new TimeSpan(0, 0, 30);
        DateTime start = DateTime.Now;
        while (DateTime.Now - start < durationToRun)
        {
            Console.WriteLine("not finished yet");
        }
        Console.WriteLine("finished");
    }
}
```

Jump statements

A jump statement can be used to transfer program control using keywords such as `break`, `continue`, `return`, and `yield`.

Using `yield`

A `yield` statement is used to create an iterator, i.e. a function that returns a sequence of values from an object implementing `IEnumerable`. Instead of using `return` to return the sequence, you use `yield return` to return individual values and `yield break` to end the sequence.

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

public class YieldSample {
  static IEnumerable<DateTime> GenerateTimes()
  {
    DateTime limit = DateTime.Now + new TimeSpan(0,0,30);
    while (DateTime.Now < limit) yield return DateTime.Now;
    yield break;
  }

  static void Main()
  {
    foreach (DateTime d in GenerateTimes())
    {
      System.Console.WriteLine(d);
```

```
            System.Console.Read();
        }
    }
}
```

Note that you define the function as returning a `System.Collections.Generic.IEnumerable` parameterized to some type, then `yield return` individual values of the parameter type. Also, note that the body of the calling foreach loop is executed in between the `yield return` statements.

Exception handling statements

An exception handling statement can be used to handle exceptions using keywords such as `throw`, `try-catch`, `try-finally`, and `try-catch-finally`.

See the Exceptions page for more information on Exception handling

8 Exceptions

The exception handling system in the C# allows the programmer to handle errors or anomalous situations in a structured manner that allows the programmer to separate the normal flow of the code from error handling logic. An exception can represent a variety of abnormal conditions, including, for example, the use of a null object reference detected by the runtime system or an invalid input string entered by a user and detected by application code. Code that detects an error condition is said to *throw* an exception and code that handles the error is said to *catch* the exception. An exception in C# is an object that encapsulates various information about the error that occurred, such as the stack trace at the point of the exception and a descriptive error message. All exception objects are instantiations of the `System.Exception` or a child class of it. There are many exception classes defined in the .NET Framework used for various purposes. Programmers may also define their own class inheriting from `System.Exception` or some other appropriate exception class from the .NET Framework, such as `ApplicationException`.

The following example demonstrates the basics of exception throwing and handling exceptions:

```
class ExceptionTest
{
     public static void Main(string[] args)
     {
          try
          {
               OrderPizza("pepperoni");
               OrderPizza("anchovies");
          }
          catch (ApplicationException e)
          {
               Console.WriteLine(e.Message);
          }
          finally
          {
               Console.WriteLine("press enter to continue...");
               Console.ReadLine();
          }
     }

     private static void OrderPizza(string topping)
     {
          if (topping != "pepperoni" && topping != "sausage")
          {
               throw new ApplicationException(
                    String.Format("Unsupported pizza topping: {0}", topping));
          }
          Console.WriteLine("one {0} pizza ordered", topping);
     }
}
```

When run, this example produces the following output:

```
one pepperoni pizza ordered
Unsupported pizza topping: anchovies
press enter to continue...
```

The `Main()` method begins by opening a **`try`** block. A **`try`** block is a block of code that may throw an exception that is to be caught and handled. Following the **`try`** block are one or more **`catch`** blocks. These blocks contain the exception handling logic. Each **`catch`** block contains an exception object declaration, similar to the way a method argument is declared, in this case, an `ApplicationException` named `e`. When an exception matching the type of the **`catch`** block is thrown, that exception object is passed in to the **`catch`** and available for it to use and even possibly re-throw. The **`try`** block calls the `OrderPizza()` method, which may throw an `ApplicationException`. The method checks the input string and, if it has an invalid value, an exception is thrown using the **`throw`** keyword. The **`throw`** is followed by the object reference representing the exception object to throw. In this case, the exception object is constructed on the spot. When the exception is thrown, control is transferred to the inner most **`catch`** block matching the exception type thrown. In this case, it is one method in the call stack higher. Lastly, the `Main()` method contains a **`finally`** block after the **`catch`** block. The **`finally`** block is optional and contains code that is to be executed regardless of whether an exception is thrown in the associated **`try`** block. In this case, the **`finally`** just prompts the user to press enter, but normally it is used to release acquired resources or perform other cleanup activities.

9 Namespaces

Namespaces are used to provide a "named space" in which your application resides. They're used especially to provide the C# compiler a context for all the named information in your program, such as variable names. Without namespaces, you wouldn't be able to make e.g. a class named Console, as .NET already use one in its System namespace. The purpose of namespaces is to solve this problem, and release thousands of names defined in the .NET Framework for your applications to use, along with making it so your application doesn't occupy names for other applications, if your application is intended to be used in conjunction with another. So namespaces exist to resolve ambiguities a compiler wouldn't otherwise be able to do.

Namespaces are easily defined in this way:

```
namespace MyApplication
{
    // The content to reside in the MyApplication namespace is placed here.
}
```

There is an entire hierarchy of namespaces provided to you by the .NET Framework, with the System namespace usually being by far the most commonly seen one. Data in a namespace is referred to by using the `.` operator, such as:

```
System.Console.WriteLine("Hello, world!");
```

This will call the `WriteLine` method that is a member of the `Console` class within the `System` namespace.

By using the `using` keyword, you explicitly tell the compiler that you'll be using a certain namespace in your program. Since the compiler would then know that, it no longer requires you to type the namespace name(s) for such declared namespaces, as you told it which namespaces it should look in if it couldn't find the data in your application.

So one can then type like this:

```
using System;

namespace MyApplication
{
  class MyClass
  {
    void ShowGreeting()
    {
        Console.WriteLine("Hello, world!"); // note how System is now not required
    }
  }
}
```

Namespaces are global, so a namespace in one C# source file, and another with the same name in another source file, will cause the compiler to treat the

different named information in these two source files as residing in the same namespace.

Nested namespaces

Normally, your entire application resides under its own special namespace, often named after your application or project name. Sometimes, companies with an entire product series decide to use nested namespaces though, where the "root" namespace can share the name of the company, and the nested namespaces the respective project names. This can be especially convenient if you're a developer who has made a library with some usual functionality that can be shared across programs. If both the library and your program shared a parent namespace, that one would then not have to be explicitly declared with the `using` keyword, and still not have to be completely typed out. If your code was open for others to use, third party developers that may use your code would additionally then see that the same company had developed the library and the program. The developer of the library and program would finally also separate all the named information in their product source codes, for fewer headaches especially if common names are used.

To make your application reside in a nested namespace, you can show this in two ways. Either like this:

```
namespace CodeWorks
{
    namespace MyApplication
    {
        // Do stuff
    }
}
```

... or like this:

```
namespace CodeWorks.MyApplication
{
    // Do stuff
}
```

Both methods are accepted, and are identical in what they do.

10 Classes

live version · discussion · edit chapter · comment · report an error

As in other object-oriented programming languages, the functionality of a C# program is implemented in one or more **classes**. The *methods* and *properties* of a class contain the code that defines how the class behaves.

C# classes support information hiding by encapsulating functionality in properties and methods and by enabling several types of polymorphism, including *subtyping polymorphism* via inheritance and *parametric polymorphism* via generics.

Several types of C# classes can be defined, including *instance* classes (*standard* classes that can be instantiated), *static* classes, and *structures*.

Classes are defined using the keyword "`class`" followed by an identifier to name the class. Instances of the class can then be created with the "`new`" keyword followed by the name of the class. The code below defines a class called `Employee` with properties `Name` and `Age` and with empty methods `GetPayCheck()` and `Work()`. It also defines a `Sample` class that instantiates and uses the `Employee` class:

```
public class Employee
{
    private string _name;
    private int _age;

    public string Name
    {
        set { _name = value; }
        get { return _name; }
    }

    public int Age
    {
        set { _age = value; }
        get { return _age; }
    }

    public void GetPayCheck()
    {
    }

    public void Work()
    {
    }
}

public class Sample
{
    public static void Main()
    {
        Employee Marissa = new Employee();
        Marissa.Work();
        Marissa.GetPayCheck();
    }
}
```

Methods

C# *methods* are class members containing code. They may have a return value and a list of parameters, as well as a *generic* type declaration. Like fields, methods can be *static* (associated with and accessed through the class) or *instance* (associated with and accessed through an object instance of the class).

Constructors

A class's *constructors* control its initialization. A constructor's code executes to initialize an instance of the class when a program requests a new object of the class's type. Constructors often set properties of their classes, but they are not restricted to doing so.

Like other methods, a constructor can have *parameters*. To create an object using a constructor with parameters, the `new` command accepts parameters. The below code defines and then instantiates multiple objects of the `Employee` class, once using the constructor without parameters and once using the version with a parameter:

```
public class Employee
{
    public Employee()
    {
        System.Console.WriteLine("Constructed without parameters");
    }

    public Employee(string text)
    {
        System.Console.WriteLine(text);
    }
}

public class Sample
{
    public static void Main()
    {
        System.Console.WriteLine("Start");
        Employee Alfred = new Employee();
        Employee Billy  = new Employee("Parameter for construction");
        System.Console.WriteLine("End");
    }
}
```

Output:

```
Start
Constructed without parameters
Parameter for construction
End
```

Finalizers

The opposite of constructors, *finalizers* define final the behavior of an object

and execute when the object is no longer in use. Although they are often used in C++ to free memory reserved by an object, they are less frequently used in C# due to the .NET Framework Garbage Collector. An object's finalizer, which takes no parameters, is called sometime after an object is no longer referenced, but the complexities of garbage collection make the specific timing of finalizers uncertain.

```
public class Employee
{
    public Employee(string text)
    {
        System.Console.WriteLine(text);
    }

    ~Employee()
    {
        System.Console.WriteLine("Finalized!");
    }

    public static void Main()
    {
        Employee Marissa = new Employee("Constructed!");
        Marissa = null;
    }
}
```

Output:

```
Constructed!
Finalized!
```

Properties

C# **properties** are class members that expose functionality of methods using the syntax of *fields*. They simplify the syntax of calling traditional *get* and *set* methods (a.k.a. *accessor* methods). Like methods, they can be *static* or *instance*.

Properties are defined in the following way:

```
public class MyClass
{
    private int integerField = 3; // Sets integerField with a default value of 3

    public int IntegerField
    {
        get {
            return integerField;  // get returns the field you specify when this property is
assigned
        }
        set {
            integerField = value; // set assigns the value assigned to the property of the field
you specify
        }
    }
}
```

The C# keyword **value** contains the value assigned to the property. After a property is defined it can be used like a variable. If you were to write some

additional code in the get and set portions of the property it would work like a method and allow you to manipulate the data before it is read or written to the variable.

```
using System;

public class MyProgram
{
    MyClass myClass = new MyClass;

    Console.WriteLine(myClass.IntegerField); // Writes 3 to the command line.
    myClass.IntegerField = 7; // Indirectly assigns 7 to the field myClass.integerField
}
```

Using properties in this way provides a clean, easy to use mechanism for protecting data.

Indexers

C# **indexers** are class members that define the behavior of the *array access* operation (e.g. `list[0]` to access the first element of `list` even when `list` is not an array).

To create an indexer, use the **this** keyword as in the following example:

```
public string this[string key]
{
   get {return coll[_key];}
   set {coll[_key] = value;}
}
```

This code will create a string indexer that returns a string value. For example, if the class was EmployeeCollection, you could write code similar to the following:

```
EmployeeCollection e = new EmployeeCollection();
.
.
.
string s = e["Jones"];
e["Smith"] = "xxx";
```

Events

C# **events** are class members that expose notifications to clients of the class.

Operator

C# **operator** definitions are class members that define or redefine the behavior of basic C# operators (called implicitly or explicitly) on instances of the class.

Structures

Structures, or *structs*, are defined with the `struct` keyword followed by an *identifier* to name the structure. They are similar to classes, but have subtle differences. *Structs* are used as lightweight versions of classes that can help reduce memory management efforts in when working with small data structures. In most situations, however, using a standard *class* is a better choice.

The principal difference between *structs* and *classes* is that *instances* of *structs* are *values* whereas *instances* of *classes* are *references*. Thus when you pass a *struct* to a function by value you get a copy of the object so changes to it are not reflected in the original because there are now two distinct objects but if you pass an instance of a class by value then there is only one instance.

The `Employee` structure below declares a `public` and a `private` field. Access to the `private` field is granted through the `public` property "`Name`":

```
struct Employee
{
    private string name;
    public int age;

    public string Name
    {
        set { name = value; }
        get { return name; }
    }
}
```

Static classes

Static classes are commonly used to implement a Singleton Pattern. All of the methods, properties, and fields of a `static` class are also `static` (like the `WriteLine()` method of the `System.Console` class) and can thus be used without instantiating the `static` class:

```
public static class Writer
{
    public static void Write()
    {
        System.Console.WriteLine("Text");
    }
}

public class Sample
{
    public static void Main()
    {
        Writer.Write();
    }
}
```

live version · discussion · edit chapter · comment · report an error

11 Encapsulation

Encapsulation is depriving of the user of a class information he does not need, and preventing him from manipulating objects in ways not intended by the designer.

A class element having *public protection level* is accessible to all code anywhere in the program. These methods and properties represent the operations allowed on the class to outside users.

Methods, data members (and other elements) with *private protection level* represent the internal state of the class (for variables), and operations which are not allowed to outside users.

For example:

```
public class Frog
{
        public void JumpLow() { Jump(1); }
        public void JumpHigh() { Jump(10); }

        private void Jump(int height) { _height += height;}

        private int _height = 0;
}
```

In this example, the public method the Frog class exposes are JumpLow and JumpHigh. Internally, they are implemented using the private Jump function that can jump to any height. This operation is not visible to an outside user, so he cannot make the frog jump 100 meters, only 10 or 1. The Jump private method is implemented by changing the value of a private data member _height, which is also not visible to an outside user. Some private data members are made visible by properties.

Protection Levels

Private

Private members are only accessible within the class itself. A method in another class, even a class derived from the class with private members cannot access the members.

Protected

Protected members can be accessed by the class itself and by any class

derived from that class.

Public

Public members can be accessed by any method in any class.

Internal

Internal members are accessible only in the same assembly and invisible outside it.

12 .NET Framework Overview

Introduction

NET was originally called NGWS(Next Generation Windows Services). .NET is a new Internet and Web based infrastructure .NET delivers software as Web Services .NET is a server centric computing model .NET will run in any browser on any platform - .Net does not run IN any browser. It is a RUNTIME language (Common Language Runtime) like the Java runtime. Silverlight does run in a browser, but has nothing to do with .Net .NET is based on the newest Web standards

.NET is built on the following Internet standards:

- HTTP, the communication protocol between Internet Applications
- XML, the format for exchanging data between Internet Applications
- SOAP, the standard format for requesting Web Services
- UDDI, the standard to search and discover Web Services

The .NET Framework is a common environment for building, deploying, and running Web Services and Web Applications.The .NET Framework contains common class libraries - like ADO.NET, ASP.NET and Windows Forms - to provide advanced standard services that can be integrated into a variety of computer systems.

In the System namespace, there are a lot of useful libraries. Let's look at a couple. If you want to start up a external program, or a webpage, you can write:

```
System.Diagnostics.Process.Start("notepad.exe");
System.Diagnostics.Process.Start("http://www.wikibooks.org");
```

You can also get information about your system in the Environment namespace:

```
Console.WriteLine("Machine name: " + System.Environment.MachineName);
Console.WriteLine(System.Environment.OSVersion.Platform.ToString() + "   "
                + System.Environment.OSVersion.Version.ToString());
```

User input

You can read a line from the user with

```
Console.ReadLine()
```

This can be directly passed as a parameter to Console.Write:

```
Console.Write ( "I'm afraid I can't do that, {0}" , Console.ReadLine() ) ;
```

which will be most effective for the input "Dave" :-)

In this case, "{0}" gets replaced with the first parameter passed to Console.Write(), which is Console.ReadLine(). "{1}" would be the next parameter etc.

13 Console Programming

Output

The example program below shows a couple ways to output text:

```
using System;
public class HelloWorld
{
   public static void Main()
   {
       Console.WriteLine("Hello World!");             // relies on "using System;"
       Console.Write("This is");
       Console.Write("... my first program!\n");
       System.Console.WriteLine("Goodbye World!");    // no "using" statement required
   }
}
```

The above code displays the following text:

```
Hello World!
This is... my first program!
Goodbye World!
```

That text is output using the `System.Console` class. The `using` statement at the top allows the compiler to find the `Console` class without specifying the `System` namespace each time it is used.

The middle lines use the `Write()` method, which does not automatically create a new line. To specify a new line, we can use the sequence backslash-n (`\n`). If for whatever reason we wanted to really show the \n character instead, we add a second backslash (`\\n`). The backslash is known as the escape character in C# because it is not treated as a normal character, but allows us to encode certain special characters (like a new line character).

Input

Input can be gathered in a similar method to outputing data using the `Read()` and `ReadLine` methods of that same `System.Console` class:

```
using System;
public class ExampleClass
{
   public static void Main()
   {
       Console.WriteLine("Greetings!  What is your name?");
       Console.Write("My name is: ");
       string name = Console.ReadLine();
       Console.WriteLine("Nice to meet you, " + name);
       Console.Read();
   }
}
```

The above program requests the user's name and displays it back. The final `Console.Read()` waits for the user to enter a key before exiting the program.

Command line arguments

Command line arguments are values that are passed to a console program before execution. For example, the Windows command prompt includes a `copy` command that takes two command line arguments. The first argument is the original file and the second is the location or name for the new copy. Custom console applications can have arguments as well.

```
using System;
public class ExampleClass
{
   public static void Main(string[] args)
   {
       Console.WriteLine("First Name: " + args[0]);
       Console.WriteLine("Last Name: " + args[1]);
       Console.Read();
   }
}
```

If the program above code is compiled to a program called *username.exe*, it can be executed from the command line using two arguments, e.g. "Bill" and "Gates":

```
C:\>username.exe Bill Gates
```

Notice how the `Main()` method above has a string array parameter. The program assumes that there will be two arguments. That assumption makes the program unsafe. If it is run without the expected number of command line arguments, it will crash when it attempts to access the missing argument. To make the program more robust, we make we can check to see if the user entered all the required arguments.

```
using System;
public class Test
{
   public static void Main(string[] args)
   {
       if(args.Length >= 1)
           Console.WriteLine(args[0]);
       if(args.Length >= 2)
           Console.WriteLine(args[1]);
   }
}
```

Try running the program with only entering your first name or no name at all. The `string.Length` property returns the total number of arguments. If no arguments are given, it will return zero.

14 Windows Forms

live version · discussion · edit chapter · comment · report an error

The `System.Windows.Forms` namespace allows us to create Windows applications easily. The `Form` class is a particularly important part of that namespace because the form is the key graphical building block Windows applications. It provides the visual frame that holds buttons, menus, icons and title bars together. Integrated development environments (IDEs) like Visual C# and SharpDevelop can help create graphical applications, but it is important to know how to do so manually:

```
using System.Windows.Forms;
public class ExampleForm : Form    // inherits from System.Windows.Forms.Form
{
   public static void Main()
   {
       ExampleForm wikibooksForm = new ExampleForm();
       wikibooksForm.Text = "I Love Wikibooks";// specify title of the form
       wikibooksForm.Width = 400;              // width of the window in pixels
       wikibooksForm.Height = 300;             // height in pixels
       Application.Run(wikibooksForm);         // display the form
   }
}
```

The example above creates a simple Window with the text "I Love Wikibooks" in the title bar. Custom form classes like the example above inherit from the `System.Windows.Forms.Form` class. Setting any of the properties `Text`, `Width`, and `Height` is optional. Your program will compile and run successfully if you comment these lines out but they allow us to add extra control to our form.

live version · discussion · edit chapter · comment · report an error

15 Inheritance

live version · discussion · edit chapter · comment · report an error

Inheritance is the ability to create a class from another class, and extending the functionality and state of the derived class.

Inheritance in C# also allows derived classes to overload methods from their parent class.

Inheritance(by Mine)

Subtyping Inheritance

The code sample below shows two classes, `Employee` and `Executive`. Employee has the following methods, `GetPayCheck` and `Work`.

We want the `Executive` class to have the same methods, but differently implemented and one extra method, `AdministerEmployee`.

Below is the creation of the first class to be derived from, `Employee`.

```
public class Employee
   {
       // we declare one method virtual so that the Executive class can
       // override it.
       public virtual void GetPayCheck()
       {
           //get paycheck logic here.
       }

       //Employee's and Executives both work, so no virtual here needed.
       public void Work()
       {
           //do work logic here.
       }
   }
```

Now, we create an `Executive` class that will override the `GetPayCheck` method.

```
public class Executive : Employee
   {
       // the override keyword indicates we want new logic behind the GetPayCheck method.
       public override void GetPayCheck()
       {
           //new getpaycheck logic here.
       }

       // the extra method is implemented.
       public void AdministerEmployee()
       {
           // manage employee logic here
       }
   }
```

You'll notice that there is no `Work` method in the `Executive` class, it is not

required, since that method is automatically added to the `Executive` class, because it derives its methods from `Employee`, which has the `Work` method.

```
static void Main(string[] args)
       {
            Employee emp = new Employee;
            Executive exec = new Executive;

            emp.Work();
            exec.Work();
            emp.GetPayCheck();
            exec.GetPayCheck();
            exec.AdministerEmployee();
       }
```

Inheritance keywords

How C# inherits from another class syntacticaly is using the ":" character.

Example. **`public class`** `Executive : Employee`

To indicate a method that can be overridden, you mark the method with **`virtual`**.

```
public virtual void Write(string text)
{
   System.Console.WriteLine("Text:{0}", text);
}
```

To override a method use the **`override`** keyword

```
public override void Write(string  text)
{
   System.Console.WriteLine(text);
}
```

16 Interfaces

An interface in C# is type definition similar to a class except that it purely represents a contract between an object and a user of the object. An interface cannot be directly instantiated as an object. No data members can be defined in an interface. Methods and properties can only be declared, not defined. For example, the following defines a simple interface:

```
interface IShape
{
  void Draw();
  double X { get; set; }
  double Y { get; set; }
}
```

A convention used in the .NET Framework (and likewise by many C# programmers) is to place an "I" at the beginning of an interface name to distinguish it from a class name. Another common interface naming convention is used when an interface declares only one key method, such Draw() in the above example. The interface name is then formed by adding the suffix "able" to the method name. So, in the above example, the interface name would be IDrawable. This convention is also used throughout the .NET Framework.

Implementing an interface is simply done by inheriting off the interface and then defining all the methods and properties declared by the interface. For example:

```
class Square : IShape
{
     private double mX, mY;

     public void Draw() { ... }

     public double X
     {
          set { mX = value; }
          get { return mX; }
     }

     public double Y
     {
          set { mY = value; }
          get { return mY; }
     }
}
```

Although a class can only inherit from one other class, it can inherit from any number of interfaces. This is simplified form of multiple inheritance supported by C#. When inheriting from a class and one or more interfaces, the base class should be provided first in the inheritance list followed by any interfaces to be implemented. For example:

```
class MyClass : Class1, Interface1, Interface2 { ... }
```

Object references can be declared using an interface type. For example, using the previous examples:

```
class MyClass
{
     static void Main()
     {
          IShape shape = new Square();
          shape.Draw();
     }
}
```

Intefaces can inherit off of any number of other interfaces but cannot inherit from classes. For example:

```
interface IRotateable
{
     void Rotate(double theta);
}


interface IDrawable : IRotateable
{
     void Draw();
}
```

Some Important Points for Interface

Access specifiers (i.e. private, internal, etc) cannot be provided for interface members. In Interface, all members are **public by default**. A class implementing an interface must define all the members declared by the interface as public. The implementing class has the option of making an implemented method virtual if it is expected to be overridden in a a child class.

In addition to methods and properties, interfaces can declare events and indexers as well.

17 Delegates and Events

Delegates

Delegates are a construct for abstracting and creating objects that reference methods and can be used to call those methods. Delegates form the basis of event handling in C#. A delegate declaration specifies a particular method signature. References to one or more methods can be added to a delegate instance. The delegate instance can then be "called" which effectively calls all the methods that have been added to the delegate instance. A simple example:

```
delegate void Procedure();

class DelegateDemo
{
    static void Method1()
    {
        Console.WriteLine("Method 1");
    }

    static void Method2()
    {
        Console.WriteLine("Method 2");
    }

    void Method3()
    {
        Console.WriteLine("Method 3");
    }

    static void Main()
    {
        Procedure someProcs = null;
        someProcs += new Procedure(DelegateDemo.Method1);
        someProcs += new Procedure(DelegateDemo.Method2);
        DelegateDemo demo = new DelegateDemo();
        someProcs += new Procedure(demo.Method3);
        someProcs();
    }
}
```

In this example, the delegate is declared by the line `delegate void Procedure();` This statement is a complete abstraction. It does not result in executable code that does any work. It merely declares a delegate type called `Procedure` which takes no arguments and returns nothing. Next, in the `Main()` method, the statement `Procedure someProcs = null;` instantiates a delegate. Something concrete has now been created. The assignment of `someProcs` to null means that it is not initially referencing any methods. The statements `someProcs += new Procedure(DelegateDemo.Method1);` and `someProcs += new Procedure(DelegateDemo.Method2);` add two static methods to the delegate instance. (Note: the class name could have been left off of `DelegateDemo.Method1` and `DelegateDemo.Method2` because the statement is occurring in the `DelegateDemo` class.) The statement `someProcs += new Procedure(demo.Method3);` adds a non-static method to the delegate instance.

For a non-static method, the method name is preceded by an object reference. When the delegate instance is called, `Method3()` is called on the object that was supplied when the method was added to the delegate instance. Finally, the statement `someProcs();` calls the delegate instance. All the methods that were added to the delegate instance are now called in the order that they were added.

Methods that have been added to a delegate instance can be removed with the `-=` operator:

```
someProcess -= new Procedure(DelegateDemo.Method1);
```

In C# 2.0, adding or removing a method reference to a delegate instance can be shortened as follows:

```
someProcess += DelegateDemo.Method1;
someProcess -= DelegateDemo.Method1;
```

Invoking a delegate instance that presently contains no method references results in a `NullReferenceException`.

Note that if a delegate declaration specifies a return type and multiple methods are added to a delegate instance, then an invocation of the delegate instance returns the return value of the last method referenced. The return values of the other methods cannot be retrieved (unless explicitly stored somewhere in addition to being returned).

Events

An event is a special kind of delegate that facilitates event-driven programming. Events are class members which cannot be called outside of the class regardless of its access specifier. So, for example, an event declared to be public would allow other classes the use of += and -= on the event, but firing the event (i.e. invoking the delegate) is only allowed in the class containing the event. A simple example:

```
delegate void ButtonClickedHandler();

class Button
{
    public event ButtonClickedHandler ButtonClicked;

    public void SimulateClick()
    {
         if (ButtonClicked != null)
         {
              ButtonClicked();
         }
    }

    ...

}
```

A method in another class can then subscribe to the event by adding one of its methods to the event delegate:

```
Button b = new Button();
b.ButtonClicked += MyHandler;
```

Even though the event is declared public, it cannot be directly fired anywhere except in the class containing the event.

Events are used extensively in GUI programming and in the System.Windows.Forms namespace.

18 Abstract Classes

An abstract class is a class that is never intended to be instantiated directly, but only to serve as a base to other classes. An abstract class should contain at least one abstract method which derived concrete classes will implement. A class should be made abstract when there are some aspects of the implementation which must be deferred to a more specific subclass.

For example, an Employee can be an abstract class if there are concrete classes that represent more specific types of Employee (e.g. SalariedEmployee, TemporaryEmployee, etc.). Although it is not possible to instantiate the Employee class directly, a program may create instances of SalariedEmployee and TemporaryEmployee, both of which inherit the behavior defined in the Employee class.

19 Partial Classes

As the name indicates, partial class definitions can be split up across multiple physical files. To the compiler, this does not make a difference as all the fragments of the partial class are grouped and the compiler treats it as a single class. One common usage of partial classes is the separation of automatically generated code from programmer written code.

Below is the example of a partial class.

Listing 1: Entire class definition in one file (file1.cs)

```
public class Node
{
    public bool Delete()
    {
    }

    public bool Create()
    {
    }
}
```

Listing 2: Class split across multiple files

(file1.cs)

```
public partial class Node
{
    public bool Delete()
    {
    }
}
```

(file2.cs)

```
public partial class Node
{
    public bool Create()
    {
    }
}
```

20 Generics

live version · discussion · edit chapter · comment · report an error

Generics is essentially the ability to have type parameters on your type. They are also called parameterized types or parametric polymorphism. The classic example is a List collection class. A List is a convenient growable array. It has a sort method, you can index into it, and so on.

Generic Interfaces

MSDN2 Entry for Generic Interfaces

Generic Classes

There are cases, when you need to create some class to manage objects of some type, without modyfing them. Without Generics, usual approach (highly simplified) to make such class would be like this:

```
public class SomeObjectContainer
{
    private object obj;

    public SomeObjectContainer(object obj)
    {
        this.obj = obj;
    }

    public object getObject()
    {
        return this.obj;
    }
}
```

And the usage of it would be:

```
class Program
{
    static void Main(string[] args)
    {
        SomeObjectContainer container = new SomeObjectContainer(25);
        SomeObjectContainer container2 = new SomeObjectContainer(5);
        Console.WriteLine((int)container.getObject() + (int)container2.getObject());

        Console.ReadKey(); // wait for user to press any key, so we could see results
    }
}
```

Notice, that we have to cast back to original data type we have chosen (in this case - **int**) every time we want to get object from such container. In such small program like this, everything is clear. But in more complicated case with more containers in different parts of program, we would have to take care that container, supposed to have **int** type in it, would not have a **string** or any other

data type. If that happens, InvalidCastException is thrown.

Additionally, if the original data type we have chosen is a **struct** type, such as **int**, we will incur a performance penalty every time we access the elements of the collection, due to the Autoboxing feature of C#.

However, we could surround every unsafe area with **try** - **catch** block, or we could create separate "container" for every data type we need, just to avoid casting. While both ways could work (and worked for many years), it is unnecessary now, because Generics offers much more elegant solution.

To make our "container" class to support any object and avoid casting, we replace every previous **object** type with some new name, in this case - T, and add <T> mark near class name to indicate that this "T" type is Generic / any type.

> Note: You can choose any name and use more than one generic type for class, i.e <genKey, genVal>

```
public class GenericObjectContainer<T>
{
    private T obj;

    public GenericObjectContainer(T obj)
    {
        this.obj = obj;
    }

    public T getObject()
    {
        return this.obj;
    }
}
```

Not a big difference, which results in simple and safe usage:

```
class Program
{
    static void Main(string[] args)
    {
        GenericObjectContainer<int> container = new GenericObjectContainer<int>(25);
        GenericObjectContainer<int> container2 = new GenericObjectContainer<int>(5);
        Console.WriteLine(container.getObject() + container2.getObject());

        Console.ReadKey(); // wait for user to press any key, so we could see results
    }
}
```

Generics ensures, that you specify type for "container" only when creating it, and after that you will be able to use only the type you specified. But now you can create containers for different object types, and avoid previously mentioned problems. In addition, this avoids the Autoboxing for **struct** types.

While this example is far from practical, it does illustrate some situations where generics are useful:

- You need to keep objects of single type in some class

- You don't need to modify objects
- You need to manipulate objects in some way
- You wish to store a "value type" (such as **int**, **short**, **string**, or any custom **struct**) in a collection class without incurring the performance penalty of Autoboxing every time you manipulate the stored elements.

live version · discussion · edit chapter · comment · report an error

21 Object Lifetime

Introduction

Different programming languages deal with issues of object lifetime in different ways and this must be accounted for when writing programs because some objects *must* be disposed of at specific moments while others can be allowed to exist until the program terminates or a garbage collector disposes of it.

If you are coming to C# from Visual Basic Classic you will have seen code like this:

```
Public Function Read(ByRef FileName) As String

  Dim oFSO As FileSystemObject
  Set oFSO = New FileSystemObject

  Dim oFile As TextStream
  Set oFile = oFSO.OpenTextFile(FileName, ForReading, False)
  Read = oFile.ReadLine

End Function
```

Note that neither *oFSO* nor *oFile* are explicitly disposed of. In Visual Basic Classic this is not necessary because both objects are declared locally. This means that the reference count goes to zero as soon as the function ends which results in calls to the *Terminate* event handlers of both objects. Those event handlers close the file and release the associated resources.

In C# this doesn't happen because the objects are not reference counted. The finalizers will not be called until the garbage collector decides to dispose of the objects. If the program uses very little memory this could be a long time.

This causes a problem because the file is held open which might prevent other processes from accessing it.

In many languages the solution is to explicitly close the file and dispose of the objects and many C# programmers do just that. However, there is a better way: use the *using* statement:

```
public read(string fileName)
{
   using (TextReader textReader = new StreamReader(filename))
   {
      return textReader.ReadLine();
   }
}
```

Behind the scenes the compiler turns the using statement into ***try..finally***

and produces this intermediate language (*IL*) code:

```
.method public hidebysig static string  Read(string FileName) cil managed
{
  // Code size       39 (0x27)
  .maxstack  5
  .locals init (class [mscorlib]System.IO.TextReader V_0,
           string V_1)
  IL_0000:  ldarg.0
  IL_0001:  newobj     instance void [mscorlib]System.IO.StreamReader::.ctor(string)
  IL_0006:  stloc.0
  .try
  {
    IL_0007:  ldloc.0
    IL_0008:  callvirt   instance string [mscorlib]System.IO.TextReader::ReadLine()
    IL_000d:  stloc.1
    IL_000e:  leave      IL_0025
    IL_0013:  leave      IL_0025
  }  // end .try
  finally
  {
    IL_0018:  ldloc.0
    IL_0019:  brfalse    IL_0024
    IL_001e:  ldloc.0
    IL_001f:  callvirt   instance void [mscorlib]System.IDisposable::Dispose()
    IL_0024:  endfinally
  }  // end handler
  IL_0025:  ldloc.1
  IL_0026:  ret
} // end of method Using::Read
```

Notice that the body of the *Read* function has been split into three parts: initialisation, try, and finally. The *finally* block includes code that was never explicitly specified in the original C# source code, namely a call to the *destructor* of the *Streamreader* instance.

See Understanding the 'using' statement in C# By TiNgZ aBrAhAm.

Resource Acquisition Is Initialisation

The application of the *using* statement in the introduction is an example of an idiom called *Resource Acquisition Is Initialisation* (RAII).

RAII is a natural technique in languages like Visual Basic Classic and C++ that have deterministic finalization but usually requires extra work to include in programs written in garbage collected languages like C# and VB.NET. The *using* statement makes it just as easy. Of course you could write the *try..finally* code out explicitly and in some cases that will still be necessary. For a thorough discussion of the RAII technique see HackCraft: The RAII Programming Idiom. Wikipedia has a brief note on the subject as well: Resource Acquisition Is Initialization.

Work in progress: add C# versions showing incorrect and correct methods with and without using. Add notes on RAII, memoization and cacheing (see OOP wikibook).

live version · discussion · edit chapter · comment · report an error

22 History & Document Notes

Wikibook History

This book was created on 2004-06-15 and was developed on the Wikibooks project by the contributors listed in the next section. The latest version may be found at http://en.wikibooks.org/wiki/C_Sharp_Programming.

PDF Information & History

This PDF was created on 2007-07-15 based on the 2007-07-14 version of the C# Programming Wikibook. A transparent copy of this document is available at Wikibooks:C Sharp Programming. The SXW source of this PDF document is available at Wikibooks:Image:C Sharp Programming.sxw. The template from which the document was created is available at Wikibooks:Image:PDF template.sxw.

23 Authors

Principal Authors

- Rodasmith (Contributions)
- Jonas Nordlund (Contributions)
- Eray (Contributions)
- Jlenthe (Contributions)
- Nercury (Contributions)
- Ripper234 (Contributions)

All Authors

Andyrewww, Arbitrary, Bacon, Bijee, Boly38, Charles Iliya Krempeaux, Chmohan, Chowmeined, Cpons, Darklama, David C Walls, Derbeth, Dethomas, Devourer09, Dm7475, Eray, Fatcat1111, Feraudyh, Fly4fun, Frank, HGatta, Hagindaz, Herbythyme, Huan086, Hyad, Jesperordrup, Jguk, Jlenthe, Jokes Free4Me, Jonas Nordlund, Kernigh, Krischik, Kwhitefoot, Luke101, Lux-fiat, Magnus Manske, Minun, Mkn, Mshonle, Nanodeath, Nercury, Northgrove, Nym, Ohms law, Orion Blastar, Panic2k4, Pcu123456789, Peachpuff, Plee, Ripper234, Rodasmith, Scorchsaber, Shall mn, Sytone, Szelee, Thambiduraip, Whiteknight, Withinfocus, Yurik, Zr40

24 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more

than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the

Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does

not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

External links

- GNU Free Documentation License (Wikipedia article on the license)
- Official GNU FDL webpage