C Sharp Programming
On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. An URI to this license is given in the list of figures on page 165. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 159. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 169, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 165. This PDF was generated by the \LaTeX typesetting software. The \LaTeX source code is included as an attachment (source.7z.txt) in this PDF file. To extract the source from the PDF file, we recommend the use of http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/ utility or clicking the paper clip attachment symbol on the lower left of your PDF Viewer, selecting Save Attachment. After extracting it from the PDF file you have to rename it to source.7z. To uncompress the resulting archive we recommend the use of http://www.7-zip.org/. The \LaTeX source itself was generated by a program written by Dirk Hünniger, which is freely available under an open source license from http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf. This distribution also contains a configured version of the pdflatex compiler with all necessary packages and fonts needed to compile the \LaTeX source included in this PDF file.
## Contents

1. **Introduction**
   1.1 Introduction ........................................... 3  
   1.2 Standard .............................................. 4  
   1.3 History ............................................... 4  
   1.4 References ........................................... 5  
   1.5 Microsoft .NET ........................................ 5  
   1.6 Mono .................................................. 5  
   1.7 Hello, World! ......................................... 6  

2. **Language Basics**
   2.1 Reasoning .............................................. 9  
   2.2 Conventions .......................................... 9  
   2.3 Example .............................................. 11  
   2.4 Statements .......................................... 12  
   2.5 Statement blocks .................................... 14  
   2.6 Comments ............................................ 14  
   2.7 Case sensitivity ..................................... 15  
   2.8 Fields, local variables, and parameters .......... 16  
   2.9 Types ............................................... 17  
   2.10 Text & variable example ............................ 22  
   2.11 Scope and extent .................................... 23  
   2.12 Arithmetic .......................................... 23  
   2.13 Logical .............................................. 25  
   2.14 Bitwise shifting .................................... 27  
   2.15 Relational .......................................... 27  
   2.16 Assignment ......................................... 28  
   2.17 Short-hand Assignment .............................. 29  
   2.18 Type information .................................... 29  
   2.19 Pointer manipulation ................................ 30  
   2.20 Overflow exception control ....................... 30  
   2.21 Others ............................................... 30  
   2.22 Enumerations ....................................... 31  
   2.23 Structs .............................................. 32  
   2.24 Arrays ............................................... 33  
   2.25 Conditional statements ............................ 34  
   2.26 Iteration statements ............................... 36  
   2.27 Jump statements .................................... 38  
   2.28 Introduction ....................................... 41  
   2.29 Overview ........................................... 42  
   2.30 Examples ............................................ 42
1 Introduction

C#1 (pronounced "See Sharp") is a multi-purpose computer programming language2 suitable for all development needs.

1.1 Introduction

Although C# is derived from the C programming language3, it has features such as garbage collection4 that allow beginners to become proficient in C# more quickly than in C5 or C++6. Similar to Java7, it is object-oriented8, comes with an extensive class library, and supports exception handling, multiple types of polymorphism9, 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 development10 projects, projects implemented by individuals or large or small teams, Internet applications, and projects with strict reliability requirements. Testing frameworks such as NUnit11 make C# amenable to test-driven development12 and thus a good language for use with Extreme Programming13 (XP). Its strong typing14 helps to prevent many programming errors that are common in weakly typed languages. Because of this similarities to other languages, it is possible to introduce C# as a language with features of C++ having the programming style of Java and the rapid application model of BASIC.15

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.

1 http://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29
2 http://en.wikipedia.org/wiki/programming%20language
4 http://en.wikipedia.org/wiki/Garbage%20collection%20computer%20science
5 http://en.wikibooks.org/wiki/Programming%3AC
6 http://en.wikibooks.org/wiki/Programming%3AC%20plus
7 http://en.wikipedia.org/wiki/Java%20programming%20language
9 http://en.wikipedia.org/wiki/polymorphism%20computer%20science
10 http://en.wikipedia.org/wiki/Rapid%20application%20development
11 http://en.wikipedia.org/wiki/NUnit
12 http://en.wikipedia.org/wiki/test-driven%20development
Because of the similarities between C# and the C family of languages, as well as Java\textsuperscript{17}, a developer with a background in object-oriented languages like C++ may find C# structure and syntax intuitive.

### 1.2 Standard

w:C Sharp (programming language)\textsuperscript{18} Microsoft\textsuperscript{19}, with Anders Hejlsberg\textsuperscript{20} as Chief Engineer, created C# as part of their .NET\textsuperscript{21} initiative and subsequently opened its specification\textsuperscript{22} via the ECMA\textsuperscript{23}. Thus, the language is open to implementation by other parties. Other implementations include Mono\textsuperscript{24} and DotGNU\textsuperscript{25}.

C# and other .NET languages rely on an implementation of the virtual machine\textsuperscript{26} specified in the Common Language Infrastructure\textsuperscript{27}, like Microsoft's Common Language Runtime\textsuperscript{28} (CLR). That virtual machine manages memory, handles object references, and performs Just-In-Time (JIT) compiling of Common Intermediate Language\textsuperscript{29} 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.

### 1.3 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.\textsuperscript{30,31,32}

\textsuperscript{17} http://en.wikipedia.org/wiki/Java%20programming%20language
\textsuperscript{18} http://en.wikipedia.org/wiki/C%20Sharp%20programming%20language
\textsuperscript{19} http://en.wikipedia.org/wiki/Microsoft
\textsuperscript{20} http://en.wikipedia.org/wiki/Anders%20Hejlsberg
\textsuperscript{21} http://en.wikipedia.org/wiki/Microsoft%20.Net
\textsuperscript{22} http://www.ecma-international.org/publications/standards/Ecma-334.htm
\textsuperscript{23} http://en.wikipedia.org/wiki/ECMA%20International
\textsuperscript{24} http://en.wikipedia.org/wiki/Mono%20development%20platform
\textsuperscript{25} http://en.wikipedia.org/wiki/DotGNU
\textsuperscript{26} http://en.wikipedia.org/wiki/virtual%20machine
\textsuperscript{27} http://en.wikipedia.org/wiki/Common%20Language%20Infrastructure
\textsuperscript{28} http://en.wikipedia.org/wiki/Common%20Language%20Runtime
\textsuperscript{29} http://en.wikipedia.org/wiki/Common%20Intermediate%20Language
\textsuperscript{31} C# Programming. Retrieved 2012-10-21 http://
1.4 References

Foreword

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.

1.5 Microsoft .NET

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.

1.6 Mono

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

1.6.1 Linux

In Linux you can use the MonoDevelop IDE, and either download from their website at: MonoDevelop Downloads, or install via apt-get or your distro's installer.

- Debian-based distros:
  
  sudo apt-get install monodevelop

- Arch Linux:
  
  sudo pacman -S mono monodevelop

References

34 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming
35 http://msdn.microsoft.com/netframework/
36 http://www.mono-project.com/Downloads
37 http://dotgnu.org/pnet.html
38 http://monodevelop.com/Download
1.6.2 Windows

You can download MonoDevelop from their website at: Mono website. Click the Windows icon, and follow the installation instructions.

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 are sold commercially. The Visual Studio C# Express Edition can be downloaded and used for free from Microsoft's website.

1.7 Hello, World!

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:

```csharp
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 hello.cs -o hello.exe`.

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

41 [http://go.microsoft.com/fwlink/?linkid=7653518](http://go.microsoft.com/fwlink/?linkid=7653518)
• On Windows, use hello.exe.
• On Linux, use mono hello.exe or ilrun hello.exe.

Alternatively, in Visual C# express, you could just hit F5 or the green play button to run the code. If you want to run without debugging, press CTRL-F5.

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.
2 Language Basics

This section will define the naming conventions that are generally accepted by the C# development community. Some companies may define naming conventions that differ from this, but that is done on an individual basis and is generally discouraged. Some of the objects discussed in this section may be beyond the reader's knowledge at this point, but this section can be referred back to later.

2.1 Reasoning

Much of the naming standards are derived from Microsoft's .NET Framework libraries. These standards have proven to make names readable and understandable "at a glance". By using the correct conventions when naming objects, you ensure that other C# programmers who read your code will easily understand what objects are without having to search your code for their definition.

2.2 Conventions

2.2.1 Namespace

Namespaces are named using Pascal Case\(^1\) (also called UpperCamelCase) with no underscores. This means the first letter of every word in the name is capitalized. For example: MyNewNamespace. Also, note that Pascal Case also denotes that acronyms of three or more letters should only have the first letter capitalized (MyXmlNamespace instead of MyXMLNamespace).

2.2.2 Assemblies

If an assembly contains only one namespace, they should use the same name. Otherwise, Assemblies should follow the normal Pascal Case format.

2.2.3 Classes and Structures

Pascal Case, no underscores or leading C, c1s, or I. Classes should not have the same name as the namespace in which they reside. Any acronyms of three or more letters should be pascal case, not all caps. Try to avoid abbreviations, and try to always use nouns.

---

\(^1\) [http://en.wikipedia.org/wiki/Pascal%20Case](http://en.wikipedia.org/wiki/Pascal%20Case)
2.2.4 Exception Classes

Follow class naming conventions, but add Exception to the end of the name. In .Net 2.0, all classes should inherit from the `System.Exception` base class, and not inherit from the `System.ApplicationException`.

2.2.5 Interfaces

Follow class naming conventions, but start the name with `I` and capitalize the letter following the `I`. Example: `IFoo` The `I` prefix helps to differentiate between Interfaces and classes and also to avoid name collisions.

2.2.6 Functions

Pascal Case, no underscores except in the event handlers. Try to avoid abbreviations. Many programmers have a nasty habit of overly abbreviating everything. This should be discouraged.

2.2.7 Properties and Public Member Variables

Pascal Case, no underscores. Try to avoid abbreviations.

2.2.8 Parameters and Procedure-level Variables

Camel Case (or `lowerCamelCase`). Try to avoid abbreviations. Camel Case is the same as Pascal case, but the first letter of the first word is lowercased.

2.2.9 Class-level Private and Protected Variables

Camel Case with a leading underscore. Always indicate `protected` or `private` in the declaration. The leading underscore is the only controversial thing in this document. The leading character helps to prevent name collisions in constructors (a parameter and a private variable having the same name).

2.2.10 Controls on Forms

Pascal Case with a prefix that identifies it as being part of the UI instead of a purely coded control (example a temporary variable). Many developers use `ui` as the prefix followed by a descriptive name such as `txtUserName` or `lblUserNickName` ("txt" stands for TextBox control and "lbl" for Label control)

Some samples are below for ASP.Net web form controls:

<table>
<thead>
<tr>
<th>Control</th>
<th>Prefix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10
### 2.2.11 Constants

Pascal Case. The use of SCREAMING\_CAPS is discouraged. This is a large change from earlier conventions. Most developers now realize that in using SCREAMING\_CAPS they betray more implementation than is necessary. A large portion of the .NET Framework Design Guidelines\(^2\) is dedicated to this discussion.

### 2.3 Example

Here is an example of a class that uses all of these naming conventions combined.

```csharp
using System;

namespace MyExampleNamespace
{
    public class Customer : IDisposable
    {
        // Class implementation
    }
}
```

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.

### 2.4 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:

```csharp
int sampleVariable; // declaring a variable
sampleVariable = 5; // assigning a value
Method(); // calling an instance method
SampleClass sampleObject = new SampleClass(); // creating a new instance of an object
sampleObject.ObjectMethod(); // calling a member function of an object

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

```csharp
using System;
namespace MyExampleNamespace
{
    public class Customer : IDisposable // Interface
    {
        private string _customerName;
        public string CustomerName
        {
            get
            {
                return _customerName;
            }
            set
            {
                _customerName = value;
                _lastUpdated = DateTime.Now;
            }
        }

        private DateTime _lastUpdated;
        public DateTime LastUpdated
        {
            get
            {
                return _lastUpdated;
            }
            private set
            {
                _lastUpdated = value;
            }
        }

        public void UpdateCustomer(string newName)
        {
        }
    }
}
```
if (!newName.Equals(CustomerName))
{
    CustomerName = newName;
}

public void Dispose()
{
    //Do nothing
}

// Submitted by krishna mali [pune]

### 2.5 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.

```csharp
private void MyMethod(int integerValue)
{
    // This block of code is the body of "MyMethod()"
    // The ,value, integer parameter is accessible to everything in the method
    int methodLevelVariable; // This variable is accessible to everything in the method
    if (integerValue == 2) // This variable is accessible to everything in the method
    {
        // methodLevelVariable is still accessible here
        int limitedVariable; // This variable is only accessible to code in the, if block
        DoSomeWork(limitedVariable);
    }
    // limitedVariable is no longer accessible here
} // Here ends the code block for the body of "MyMethod()".

//submitted by krishna
```

### 2.6 Comments

*Comments* allow inline documentation of source code. The C# compiler ignores comments. These 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 is another style of a comment.
It allows multiple lines.
*/

XML Documentation-line comments

These comments are used to generate XML documentation. Single-line and multiple-line styles can be used. The single-line style, where each line of the comment begins with //, is more common than the multiple-line style delimited by /** and */.

// <summary> documentation here </summary>
// <remarks>
// This uses single-line style XML Documentation comments.
// </remarks>

/**
 * <summary> documentation here </summary>
 * <remarks>
 * This uses multiple-line style XML Documentation comments.
 * </remarks>
 */

2.7 Case sensitivity

C# is case-sensitive\(^6\), including its variable and method names.

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

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

For example, C# defines a class `Console` to handle most operations with the console window. Writing the following code would result in a compiler error unless an object named `console` had been previously defined.

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

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

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

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 a value for later use:

```csharp
string name = "Dr. Jones";
Console.WriteLine("Good morning " + name);
```

In this example, 'name' is the identifier and 'Dr. Jones' is the value that we bound to it. Also, 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. In the above example, we stored 'Dr. Jones' into a variable of the type `string`. This is a legal statement. However, if we had said `int name = "Dr. Jones"`, the compiler would have thrown an error telling us that you cannot implicitly convert between `int` and `string`. There are methods for doing this, but we will talk about them later.

### 2.8 Fields, local variables, and parameters

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

#### 2.8.1 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).

#### 2.8.2 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 on (or referenced from) the stack. They thus have both a scope and an extent of the method or statement block that declares them.

---

7 [http://ko.wikibooks.org/wiki/C%20%C0%E4%D5%04%20%D5%04%B8%5C%AD%F8%B7%98%BC%0D%2F%BE%38%2C%95](http://ko.wikibooks.org/wiki/C%20%C0%E4%D5%04%20%D5%04%B8%5C%AD%F8%B7%98%BC%0D%2F%BE%38%2C%95)


10 [http://en.wikipedia.org/wiki/object%20%28computer%20science%29](http://en.wikipedia.org/wiki/object%20%28computer%20science%29)
2.8.3 Parameter

Parameters are variables associated with a method.

An in parameter may either have its value passed in from the caller to the method's environment, so that changes to the parameter by the method do not affect the value of the caller's variable, or passed in by reference, so that changes to the variables will affect the value of the caller's variable. Value types (int, double, string) are passed in "by value" while reference types (objects) are passed in "by reference." Since this is the default for the C# compiler, it is not necessary to use .

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 caller'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.

// Each pair of lines is what the definition of a method and a call of a
// method with each of the parameters types would look like.
// In param:
void MethodOne(int param1) // definition
    MethodOne(variable); // call

// Out param:
void MethodTwo(out string message) // definition
    MethodTwo(out variable); // call

// Reference param;
void MethodThree(ref int someFlag) // definition
    MethodThree(ref theFlag) // call

// Params
void MethodFour(params string[] names) // definition

2.9 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.

There is a fundamental difference between value types and reference types: Value types are allocated on the stack, whereas reference types are allocated on the heap.
2.9.1 Value types

The value types in the .NET framework are usually small, frequently used types. The benefit of using them is that the type requires very little resources to get up and running by the CLR. Value types do not require memory to be allocated on the heap and therefore will not cause garbage collection. However, in order to be useful, the value types (or types derived from it) should remain small - ideally below 16 bytes of data. If you choose to make your value type bigger, it is recommended that you do not pass it to methods (which can require a copy of all its fields), or return it from methods.

Although this sounds like a useful type to have, it does have some flaws, which need to be understood when using it.

- Value types are always copied (intrinsically) before being passed to a method. Changes to this new object will not be reflected back in the original object passed into the method.
- Value types do not /need/ you to call their constructor. They are automatically initialized.
- Value types always initialize their fields to 0 or null.
- Value types can NEVER be assigned a value of null (but can using Nullable types)
- Value types sometimes need to be boxed (wrapped inside an object), allowing their values to be used like objects.

2.9.2 Reference types

Reference types are managed very differently by the CLR. All reference types consist of two parts: A pointer to the heap (which contains the object), and the object itself. Reference types are slightly heavier weight because of the management behind the scenes needed to keep track of them. However, this is a minor price to pay for the flexibility and speed gains from passing a pointer around, rather than copying values to/from methods.

When an object is initialized, by use of the constructor, and is of a reference type, the CLR must perform four operations:

1. The CLR calculates the amount of memory required to hold the object on the heap.
2. The CLR inserts the data into the newly created memory space.
3. The CLR marks where the end of the space lies, so that the next object can be placed there.
4. The CLR returns a reference to the newly created space.

This occurs every single time an object is created. However the assumption is that there is infinite memory, therefore some maintenance needs to take place - and that's where the garbage collector comes in.

2.9.3 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.Int32, 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). This is also known as casting.
object boxedInteger = 97;
int unboxedInteger = (int) boxedInteger;

Boxing and casting are, however, not type-safe: the compiler won't generate an error if the programmer mixes up the types. In the following short example the mistake is quite obvious, but in complex programs it may be very difficult to spot. Avoid boxing, if possible.

object getInteger = "97";
int anInteger = (int) getInteger; // No compile-time error. The program will crash, however.

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

### Integers

<table>
<thead>
<tr>
<th>C# Alias</th>
<th>.NET Type</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbyte</td>
<td>System.SByte</td>
<td>8</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>byte</td>
<td>System.Byte</td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>System.Int16</td>
<td>16</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>ushort</td>
<td>System.UInt16</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>char</td>
<td>System.Char</td>
<td>16</td>
<td>A unicode character of code 0 to 65,535</td>
</tr>
<tr>
<td>int</td>
<td>System.Int32</td>
<td>32</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>uint</td>
<td>System.UInt32</td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>ulong</td>
<td>System.UInt64</td>
<td>64</td>
<td>0 to 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

### Floating-point

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

### Other predefined types

<table>
<thead>
<tr>
<th>C# Alias</th>
<th>.NET Type</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
</table>
| bool     | System.Boolean   | 32          | true or false, which aren't related to any integer in C#.

20
<table>
<thead>
<tr>
<th>C# Alias</th>
<th>.NET Type</th>
<th>Size (bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>System.Object</td>
<td>32/64</td>
<td>Platform dependant (a pointer to an object).</td>
</tr>
<tr>
<td>string</td>
<td>System.String</td>
<td>16*length</td>
<td>A unicode string with no special upper bound.</td>
</tr>
</tbody>
</table>

2.9.4 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\(^\text{11}\) are declared with the `class` keyword.

**Arrays**

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

```csharp
string[] a_str;
```

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

```csharp
a_str = new string[5];
```

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

```csharp
string[] a_str = 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:

```csharp
static void swap (int[] a_iArray, int iI, int iJ)
{
    int iTemp = a_iArray[iI];
    a_iArray[iI] = a_iArray[iJ];
    a_iArray[iJ] = iTemp;
}
```

It is possible to determine the array size during runtime. The following example assigns the loop counter to the unsigned short array elements:

```csharp
ushort[] a_usNumbers = new ushort[234];
[…]
for (ushort us = 0; us < a_usNumbers.Length; us++)
{
    a_usNumbers = us;
}
```

\(^\text{11}\) Chapter 3.1 on page 50
Since C# 2.0, it is possible to have arrays also inside of structures\textsuperscript{12}.

## 2.10 Text & variable example

```csharp
using System;

namespace Login
{
    class Username_Password
    {
        public static void Main()
        {
            string username, password;
            Console.Write("Enter username: ");
            username = Console.ReadLine();
            Console.Write("Enter password: ");
            password = Console.ReadLine();

            if (username == "SomePerson" && password == "SomePassword")
            {
                Console.WriteLine("Access Granted.");
            }
            else if (username != "SomePerson" && password == "SomePassword")
            {
                Console.WriteLine("The username is wrong.");
            }
            else if (username == "SomePerson" && password != "SomePassword")
            {
                Console.WriteLine("The password is wrong.");
            }
            else
            {
                Console.WriteLine("Access Denied.");
            }
        }
    }
}
```

### 2.10.1 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 \textit{implicit} (i.e. an explicit \textit{cast} is not required).

\textsuperscript{12} Chapter 3.9 on page 59
**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.

**2.11 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.

**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.

Operators can be grouped by their arity\(^\text{15}\) as unary\(^\text{16}\), binary\(^\text{17}\).

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

**2.12 Arithmetic**

The following arithmetic operators operate on numeric operands (arguments \(a\) and \(b\) in the "sample usage" below).
<table>
<thead>
<tr>
<th>Sample usage</th>
<th>Read</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>a plus b</td>
<td>binary</td>
<td>+ returns the sum(^{18}) of its arguments.</td>
</tr>
<tr>
<td>a - b</td>
<td>a minus b</td>
<td>binary</td>
<td>- returns the difference(^{19}) between its arguments.</td>
</tr>
<tr>
<td>a•b</td>
<td>a times b</td>
<td>binary</td>
<td>* returns the multiplicative product(^{20}) of its arguments.</td>
</tr>
<tr>
<td>a/b</td>
<td>a divided by b</td>
<td>binary</td>
<td>/ returns the quotient(^{21}) of its arguments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If both of its operators are integers, it obtains that quotient using integer division (i.e. it drops any resulting remainder).</td>
</tr>
<tr>
<td>a%b</td>
<td>a mod b</td>
<td>binary</td>
<td>% operates only on integer arguments. It returns the remainder(^{22}) of integer division of those arguments. (See modular arithmetic(^{23})).</td>
</tr>
<tr>
<td>a++</td>
<td>a plus plus or Postincrement a</td>
<td>unary</td>
<td>++ 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.</td>
</tr>
</tbody>
</table>

---

18 http://en.wikipedia.org/wiki/addition
21 http://en.wikipedia.org/wiki/division%20%28mathematics%29
22 http://en.wikipedia.org/wiki/remainder
23 http://en.wikipedia.org/wiki/modular%20arithmetic
2.13 Logical

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

<table>
<thead>
<tr>
<th>Sample usage</th>
<th>Read</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>++a</td>
<td>plus plus a or Preincrement a</td>
<td>unary</td>
<td>++ 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.</td>
</tr>
<tr>
<td>a--</td>
<td>a minus minus or Postdecrement a</td>
<td>unary</td>
<td>-- 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.</td>
</tr>
<tr>
<td>--a</td>
<td>minus minus a or Predecrement a</td>
<td>unary</td>
<td>-- 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.</td>
</tr>
</tbody>
</table>

### Sample usage

<table>
<thead>
<tr>
<th>Read</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \land b)</td>
<td>binary</td>
<td>(\land) 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 &amp; operator. This is an example of Short Circuit Evaluation(^{25}).</td>
</tr>
<tr>
<td>(a</td>
<td>b)</td>
<td>binary</td>
</tr>
<tr>
<td>(a</td>
<td></td>
<td>b)</td>
</tr>
<tr>
<td>(a \oplus b)</td>
<td>binary</td>
<td>(\oplus) returns the exclusive or(^{28}) (&quot;XOR&quot;) of their results. If the operands are integral, the exclusive or is performed bitwise.</td>
</tr>
<tr>
<td>!(a)</td>
<td>unary</td>
<td>(!) operates on a boolean operand only. It evaluates its operand and returns the negation(^{29}) (&quot;NOT&quot;) of the result. That is, it returns (true) if (a) evaluates to (false) and it returns (false) if (a) evaluates to (true).</td>
</tr>
</tbody>
</table>

\(^{26}\) [http://en.wikipedia.org/wiki/Logical%20disjunction](http://en.wikipedia.org/wiki/Logical%20disjunction)  
\(^{28}\) [http://en.wikipedia.org/wiki/exclusive%20or](http://en.wikipedia.org/wiki/exclusive%20or)  
2.14 Bitwise shifting

<table>
<thead>
<tr>
<th>Sample usage</th>
<th>Read</th>
<th>Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \ll b)</td>
<td>(a \text{ left shift } b)</td>
<td>binary</td>
<td>(\ll) 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.</td>
</tr>
<tr>
<td>(a \gg b)</td>
<td>(a \text{ right shift } b)</td>
<td>binary</td>
<td>(\gg) 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.</td>
</tr>
</tbody>
</table>

2.15 Relational

The binary relational operators \(==\), \(!=\), \(<\), \(>\), \(<=\), and \(>=\) are used for relational operations and for type comparisons.
Sample usage

<table>
<thead>
<tr>
<th>Read</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>a is equal to b</td>
</tr>
<tr>
<td>a != b</td>
<td>a is not equal to b</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>a is less than b</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>a is greater than b</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>a is less than or equal to b</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>a is greater than or equal to b</td>
</tr>
</tbody>
</table>

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.

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.

The operator < operates on integral types. It returns true, if a is less than b, false otherwise.

The operator > operates on integral types. It returns true, if a is greater than b, false otherwise.

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

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

2.16 Assignment

The assignment operators are binary. The most basic is the operator =. Not surprisingly, it assigns the value (or reference) 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 that 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**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Read</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b</code></td>
<td><code>a equals (or set to) b</code></td>
<td>The operator <code>=</code> evaluates its second argument and then assigns the results to (the l-value indicated by) its first argument.</td>
</tr>
</tbody>
</table>

**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`.

### 2.17 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**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Read</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a += b</code></td>
<td><code>a plus equals (or increment by) b</code></td>
<td>Equivalent to <code>a = a + b</code>.</td>
</tr>
<tr>
<td><code>a -= b</code></td>
<td><code>a minus equals (or decrement by) b</code></td>
<td>Equivalent to <code>a = a - b</code>.</td>
</tr>
<tr>
<td><code>a *= b</code></td>
<td><code>a multiply equals (or multiplied by) b</code></td>
<td>Equivalent to <code>a = a*b</code>.</td>
</tr>
</tbody>
</table>

**a /= b**

- `a divide equals (or divided by) b` |
- `b` |

Equivalent to `a = a/b`. |

**a %= b**

- `a mod equals b` |
- `a` |

Equivalent to `a = a%b`. |

**a &= b**

- `a and equals b` |
- `a` |

Equivalent to `a = a&b`. |

**a |= b**

- `a or equals b` |
- `a` |

Equivalent to `a = a|b`. |

**a ^= b**

- `a xor equals b` |
- `a` |

Equivalent to `a = aˆb`. |

**a <<= b**

- `a left-shift equals b` |
- `a` |

Equivalent to `a = a << b`. |

**a >>= b**

- `a right-shift equals b` |
- `a` |

Equivalent to `a = a >> b`. |

### 2.18 Type information

**Expression**

- `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 cast 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.
### Expression Explanation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>typeof(T)</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 2.19 Pointer manipulation

**NOTE:** Most C# developers agree that direct manipulation and use of pointers is not recommended in C#. The language has many built-in classes to allow you to do almost any operation you want. C# was built with memory-management in mind and the creation and use of pointers is greatly disruptive to this end. This speaks to the declaration of pointers and the use of pointer notation, not arrays. In fact, a program may only be compiled in "unsafe mode", if it uses pointers.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>*a</td>
<td>Indirection operator. Allows access the object being pointed.</td>
</tr>
<tr>
<td>a-&gt;member</td>
<td>Similar to the . operator. Allows access to members of classes and structs being pointed.</td>
</tr>
<tr>
<td>a[]</td>
<td>Used to index a pointer.</td>
</tr>
<tr>
<td>&amp;a</td>
<td>References the address of the pointer.</td>
</tr>
<tr>
<td>stackalloc</td>
<td>allocates memory on the stack.</td>
</tr>
<tr>
<td>fixed</td>
<td>Temporarily fixes a variable in order that its address may be found.</td>
</tr>
</tbody>
</table>

### 2.20 Overflow exception control

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>checked(a)</td>
<td>uses overflow checking on value a</td>
</tr>
<tr>
<td>unchecked(a)</td>
<td>avoids overflow checking on value a</td>
</tr>
</tbody>
</table>

### 2.21 Others

<table>
<thead>
<tr>
<th>Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.b</td>
<td>accesses member b of type or namespace a</td>
</tr>
<tr>
<td>a[b]</td>
<td>the value of index b in a</td>
</tr>
<tr>
<td>(a)b</td>
<td>casts the value b to type a</td>
</tr>
<tr>
<td>new a</td>
<td>creates an object of type a</td>
</tr>
<tr>
<td>a + b</td>
<td>if a and b are strings, concatenates a and b. If any addend is null, the empty string is used instead. If one addend is a string and the other one is a non-string object, ToString() is called on that object before concatenation.</td>
</tr>
<tr>
<td>a + b</td>
<td>if a and b are delegates, performs delegate concatenation</td>
</tr>
<tr>
<td>a ? b : c</td>
<td>if a is true, returns the value of b, otherwise c</td>
</tr>
<tr>
<td>a ?? b</td>
<td>if a is null, returns b, otherwise returns a</td>
</tr>
<tr>
<td>@&quot;a&quot;</td>
<td>verbatim text, i.e., escape characters are ignored</td>
</tr>
</tbody>
</table>
There are various ways of grouping sets of data together in C#.

### 2.22 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:

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

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

```csharp
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 (note that the variable must be type cast in order to access the base type):

```csharp
enum Age { Infant = 0, Teenager = 13, Adult = 18; }
Age myAge = Age.Teenager; Console.WriteLine("You become a teenager at an age of {0}.", (int)myAge);
```

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:

```csharp
enum CardSuit : byte { Hearts, Diamonds, Spades, Clubs; }
```
The enumeration type is also helpful, if you need to output the value. By calling the `ToString()` method on the enumeration, will output the enumerations name (e.g. `CardSuit.Hearts.ToString()` will output "Hearts").

### 2.23 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.
- *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.
- *Structs* always have a default constructor, even if you don't want one. Classes allow you to hide the constructor away by using the 'private' modifier, whereas structures *must* have one.

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

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

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

```csharp
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:

```csharp
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;
    }
}
```
Arrays

```csharp
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);
    }
}
```

There is also an alternative syntax for initializing structs:

```csharp
struct Person
{
    public string Name;
    public int Height;
    public string Occupation;
}
```

```csharp
public class StructWikiBookSample2
{
    public static void Main()
    {
        Person john = new Person { Name = "John", Height = 182,
Occupation = "Programmer" };  
    }
}
```

`Structs` are really only used for performance reasons or, if you intend to reference it by value. Structs work best when holding a total equal to or less than 16 bytes of data. If in doubt, use classes. Data structures

### 2.24 Arrays

Arrays represent a set of items all belonging to the same type. The declaration itself may use a variable or a constant to define the length of the array. However, an array has a set length and it cannot be changed after declaration.

```csharp
// an array whose length is defined with a constant
int[] integers = new int[20];

int length = 0;
System.Console.Write("How long should the array be? ");
length = int.Parse(System.Console.ReadLine());
// an array whose length is defined with a variable
// this array still can,t change length after declaration
double[] doubles = new double[length];
```

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

A conditional statement can decide something using keywords such as `if`, `switch`.

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.

### 2.25 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#.

#### 2.25.1 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:

```csharp
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:

```csharp
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!");
            }
    }
}
```
2.25.2 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 (that 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:

```csharp
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:
        // falls through
    case 4:
        // falls through
    case 5:
        // falls through
    case 6:
        // falls through
    case 7:
        // falls through
    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:

```csharp
switch (aircraftIdent)
{
    case "C-FESO":
        Console.WriteLine("Rans S6S Coyote");
        break;
    case "C-GJIS":
        Console.WriteLine("Rans S12XL Airaile");
        break;
    default:
        Console.WriteLine("Unknown aircraft");
        break;
}
```
2.26 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#.

2.26.1 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:

\[
\text{do...while-loop ::= 'do' body 'while' '(* condition *)'}
\]

\[
\text{condition ::= boolean-expression}
\]

\[
\text{body ::= statement-or-statement-block}
\]

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 run, the body executes again. When the condition evaluates to false, the do...while loop ends.

```csharp
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.

2.26.2 for loop

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

\[
\text{for-loop ::= 'for' '(* initialization ;' condition ';' iteration ')'} \text{ body}
\]

\[
\text{initialization ::= variable-declaration | list-of-statements}
\]

\[
\text{condition ::= boolean-expression}
\]

\[
\text{iteration ::= list-of-statements}
\]

\[
\text{body ::= statement-or-statement-block}
\]
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.

```csharp
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.

### 2.26.3 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.

```csharp
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.

### 2.26.4 while loop

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

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.

```csharp
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");
    }
}
```

2.27 Jump statements

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

2.27.1 break

A break statement is used to exit from a case in a switch statement and also used to exit from for, foreach, while, do.....while loops that will switch the control to the statement immediately after the end of the loop.

```csharp
using System;

namespace JumpSample
{
    public class Entry
    {
        static void Main(string[] args)
        {
            int i;
            for (i = 0; i < 10; i++) // see the comparison, i < 10
            {
                if (i >= 3)
                {
                    break;
                    // Not run over the code, and get out of loop.
                    // Note: The rest of code will not be executed,
                    // & it leaves the loop instantly
                }
            }
        }
    }
}
```
2.27.2 continue

The `continue` keyword transfers program control just before the end of a loop. The condition for the loop is then checked, and if it is met, the loop performs another iteration.

```csharp
using System;
namespace JumpSample
{
    public class Entry
    {
        static void Main(string[] args)
        {
            int OneExternCounter = 0;
            for (int i = 0; i < 10; i++)
            {
                if (i >= 5)
                {
                    continue; // Not run over the code, and return to the beginning
                }
                OneExternCounter += 1;
            }
            // Here check the value of OneExternCounter, it will be 5, not 10.
            Console.WriteLine("The value of OneExternCounter: {0}", OneExternCounter);
        }
    }
}
```

2.27.3 return

The `return` keyword identifies the return value for the function or method (if any), and transfers control to the end of the function.

```csharp
namespace JumpSample
{
    public class Entry
    {
        static int Fun()
        {
            int a = 3;
            return a; // the code terminate here
            a = 9;  // here is a block that will not be executed
        }
        static void Main(string[] args)
        {
```
2.27.4 yield

The *yield* keyword is used to define an iterator block that produces values for an enumerator. It is typically used within a method implementation of the `IEnumerable` interface as an easy way to create an iterator. It is written in the following forms:

\[
\text{yield} ::= \text{"yield" "return" expression} \\
\text{yield} ::= \text{"yield" "break"}
\]

The following example shows the usage of the yield keyword inside the method `MyCounter`. This method defines an iterator block, and will return an enumerator object that generates the value of a counter from zero to `stop`, incrementing by `step` for each value generated.

```csharp
using System;
using System.Collections;

public class YieldSample
{
    public IEnumerable MyCounter(int stop, int step)
    {
        int i;
        for (i = 0; i < stop; i += step)
        {
            yield return i;
        }
    }

    static void Main()
    {
        foreach (int j in MyCounter(10, 2))
        {
            Console.WriteLine("{0} ", j);
        }
        // Will display 0 2 4 6 8
    }
}
```

2.27.5 throw

The *throw* keyword throws an exception. If it is located within a try block, it will transfer the control to a catch block that matches the exception - otherwise, it will check if any calling functions are contained within the matching catch block and transfer execution there. If no functions contain a catch block, the program may terminate because of an unhandled exception.

```csharp
namespace ExceptionSample
{
    public class Warrior
    {
    }
}
```
private string Name { get; set; }

public Warrior(string name)
{
    if (name == "Piccolo")
    {
        throw new Exception("Piccolo can't battle!");
    }
}

public class Entry
{
    static void Main(string[] args)
    {
        try
        {
            Warrior a = new Warrior("Goku");
            Warrior b = new Warrior("Vegeta");
            Warrior c = new Warrior("Piccolo"); // exception here!
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

Exceptions and the throw statement are described in greater detail in the Exceptions\textsuperscript{35} chapter.

\textsuperscript{35} Chapter 2.27.5 on page 41
\textsuperscript{36} \url{http://ko.wikibooks.org/wiki/C%20%C0%E4%D5%04%20%5C%AD%F8%B7%98%BC%00%2F%5C31C%5E%4%BB%38}
by a user and detected by application code, user requesting to withdraw more amount than
the account balance(business rule).

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.

Microsoft recommendations prior to version 2.0 recommended that a developer inherit from
the `ApplicationException` exception class. After 2.0 was released, this recommendation
was made obsolete and users should inherit from the `Exception` class

2.29 Overview

There are three code definitions for exception handling. These are:

- **try/catch** - Do something and catch an error, if it should occur.
- **try/catch/finally** - Do something and catch an error if it should occur, but always do
  the finally.
- **try/finally** - Do something, but always do the finally. Any exception that occurs, will be thrown after finally.

Exceptions are caught from most specific, to least specific. So for example, if you try and
access a file that does not exist, the CLR would look for exceptions in the following order:

- `FileNotFoundException`
- `IOException` (base class of `FileNotFoundException`)
- `SystemException` (base class of `IOException`)
- `Exception` (base class of `SystemException`)

If the exception being thrown does not derive or is not in the list of exceptions to catch, it
is thrown up the call stack.

Below are some examples of the different types of exceptions

2.30 Examples

2.30.1 try/catch

The try/catch performs an operation and should an error occur, will transfer control to
the catch block, should there be a valid section to be caught by:

37 [http://blogs.msdn.com/fxcop/archive/2006/04/05/569569.aspx](http://blogs.msdn.com/fxcop/archive/2006/04/05/569569.aspx) ApplicationException made obsolete

42
class ExceptionTest
{
    public static void Main(string[] args)
    {
        try
        {
            Console.WriteLine(args[0]);
            Console.WriteLine(args[1]);
            Console.WriteLine(args[2]);
            Console.WriteLine(args[3]);
            Console.WriteLine(args[4]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}

Here is an example with multiple catches:

class ExceptionTest
{
    public static void Main(string[] args)
    {
        try
        {
            string fileContents = new StreamReader(@"C:\log.txt").ReadToEnd();
        }
        catch (UnauthorizedAccessException e) // Access problems
        {
            Console.WriteLine(e.Message);
        }
        catch (FileNotFoundException e) // File does not exist
        {
            Console.WriteLine(e.Message);
        }
        catch (IOException e) // Some other IO problem.
        {
            Console.WriteLine(e.Message);
        }
    }
}

In all catch statements you may omit the type of exception and the exception variable name:

try
{
    int number = 1/0;
}
catch (DivideByZeroException)
{
    // DivideByZeroException
}
catch
{
    // some other exception
}
2.30.2 try/catch/finally

Catching the problem is a good idea, but it can sometimes leave your program in an invalid state. For example, if you open a connection to a database, an error occurs and you throw an exception. Where would you close the connection? In both the try AND exception blocks? Well, problems may occur before the close is carried out.

Therefore, the finally statement allows you to cater for the "in all cases do this" circumstance. See the example below:

```csharp
using System;

class ExceptionTest
{
    public static void Main(string[] args)
    {
        SqlConnection sqlConn = null;
        try
        {
            sqlConn = new SqlConnection( /*Connection here*/ );
            sqlConn.Open();
            // Various DB things
            // Notice you do not need to explicitly close the connection, as .Dispose() does this for you.
        }
        catch (SqlException e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            if (sqlConn != null && sqlConn.State != ConnectionState.Closed)
            {
                sqlConn.Dispose();
            }
        }
    }
}
```

Second Example

```csharp
using System;

public class exception
{
    public double num1, num2, result;
    public void add()
    {
        try
        {
            Console.WriteLine("enter your number");
            num1 = Convert.ToInt32(Console.ReadLine());
            num2 = Convert.ToInt32(Console.ReadLine());
            result = num1/num2;
        }
        catch(DivideByZeroException e) //FormatException
        {
            Console.WriteLine("(0)", e.Message);
        }
        catch(FormatException ex)
        {
```
Re-throwing exceptions

```csharp
{    Console.WriteLine("0",ex.Message);
} finally {    Console.WriteLine("turn over"); }

public void display() {    Console.WriteLine("The Result is: 0",result);
} public static void Main() {    exception ex = new exception();    ex.add();    ex.display(); }

} Notice that the SqlConnection object is declared outside of the try/catch/finally. The reason is that anything declared in the try/catch cannot be seen by the finally. By declaring it in the previous scope, the finally block is able to access it.

2.30.3 try/finally

The try/finally block allows you to do the same as above, but instead errors that are thrown are dealt with by the catch (if possible) and then thrown up the call stack.

class ExceptionTest {    public static void Main(string[] args) {        SqlConnection sqlConn = null;        try {            SqlConnection sqlConn = new SqlConnection ("//Connection here/");            sqlConn.Open();            // Various DB bits        } finally {            if (sqlConn != null && sqlConn.State != ConnectionState.Closed)            {                sqlConn.Dispose();            }
```

2.31 Re-throwing exceptions

Sometimes it is better to throw the error up the call stack for two reasons.

1. It is not something you would expect to happen.
2. You are placing extra information into the exception, to help diagnosis.

### 2.31.1 How not to throw exceptions

Some developers write empty `try/catch` statements like this:

```csharp
try
{
    // Do something
}
catch (Exception ex)
{
    // Ignore this here
}
```

This approach is not recommended. You are swallowing the error and continuing on. If this exception was an `OutOfMemoryException` or a `NullReferenceException`, it would not be wise to continue. Therefore you should always catch what you would expect to occur, and throw everything else.

Below is another example of how to incorrectly catch exceptions:

```csharp
/* Read the config file, and return the integer value. If it does not exist, then this is a problem! */
try
{
    string value = ConfigurationManager.AppSettings["Timeout"];
    if (value == null)
        throw new ConfigurationErrorsException("Timeout value is not in the configuration file.");
}
catch (Exception ex)
{
    // Do nothing!
}
```

As you can see, the `ConfigurationErrorsException` will be caught by the `catch (Exception)` block, but it is being ignored completely! This is bad programming as you are ignoring the error.

Some developers believe you should also use:

```csharp
try
{
    ...
}
catch (Exception ex)
{
    throw ex;
}
```

This is incorrect. What is happening is that the CLR will now think that the `throw ex;` statement is the source of the problem, when the problem is actually in the try section. Therefore never re-throw in this way.
2.31.2 How to catch exceptions

A better approach would be:

```csharp
/* Read the config file, and return the integer value. If it does not exist, then this is a problem! */

try
{
    string value = ConfigurationManager.AppSettings["Timeout"];  
    if (value == null)
        throw new ConfigurationErrorsException("Timeout value is not in the configuration file.");
} catch (Exception ex )
{
    throw;  // <-- Throw the existing problem!
}
```

The `throw;` keyword means preserve the exception information and throw it up the call stack.

2.31.3 Extra information within exceptions

An alternative is to give extra information (maybe local variable information) in addition to the exception. In this case, you wrap the exception within another. You usually use an exception that is as specific to the problem as possible, or create your own, if you cannot find out that is not specific enough (or if there is extra information you would wish to include).

```csharp
public OrderItem LoadItem(string itemNumber)
{
    DataTable dt = null;
    try
    {
        if (itemNumber == null)
            throw new ArgumentNullException("Item Number cannot be null","itemNumber");
        DataTable dt = DataAccess.OrderItem.Load(itemNumber);
        if (dt.Rows == 0)
            return null;
        else if (dt.Rows > 1)
            throw new DuplicateDataException( "Multiple items map to this item.",itemNumber, dt);
        OrderItem item = OrderItem.CreateInstanceFromDataRow(dt.Rows[0]);
        if (item == null)
            throw new ErrorLoadingException("Error loading Item " + itemNumber, itemNumber, dt.Rows[0]);
    } catch (DuplicateDataException dde)
    {
        throw new ErrorLoadingException("OrderItem.LoadItem failed with Item ");
    }
    return item;
}
```
itemNumber, dde); // <-- Include dde (as the InnerException)
parameter
}
catch (Exception ex)
{
    throw; // <-- We aren't expecting any other problems, so
    throw them if they occur.
}

2.31.4 References
3 Classes

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 uses 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:

```csharp
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:

```csharp
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:

```csharp
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.

### 3.1 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:

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

... or like this:

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

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

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\(^2\) by encapsulating\(^3\) functionality in properties and methods and by enabling several types of polymorphism\(^4\), including *subtyping polymorphism* via inheritance\(^5\) and *parametric polymorphism* via generics\(^6\).

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:

```csharp
public class Employee
{
    private int _Age;
    private string _Name;

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

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

    public void GetPayCheck()
    {
    }

    public void Work()
    {
    }
}

public class Sample
{
    public static void Main()
    {
        Employee marissa = new Employee();

        marissa.Work();
        marissa.GetPayCheck();
    }
}
```

\(^2\) http://en.wikipedia.org/wiki/information%20hiding

\(^3\) Chapter 3.14.2 on page 67

\(^4\) http://en.wikipedia.org/wiki/polymorphism%20%28computer%20science%29

\(^5\) Chapter 4 on page 73

\(^6\) http://en.wikipedia.org/wiki/generic%20programming
3.2 Methods

*C# methods* are class members containing code. They may have a return value and a list of parameters\(^7\), 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).

3.3 Constructors of classes

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 code below defines and then instantiates multiple objects of the *Employee* class, once using the constructor without parameters and once using the version with a parameter:

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

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

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
```

Constructors can call each other:

\(^7\) Chapter 2.8.3 on page 17
public class Employee
{
    public Employee(string strText, int iNumber)
    {
        ...
    }

    public Employee(string strText)
    : this(strText, 1234) // calls the above constructor with
    user-specified text and the default number
    {
    }

    public Employee()
    : this("default text") // calls the above constructor with
    the default text
    {
    }
}

3.4 Finalizers (Destructors)

The opposite of constructors, finalizers define the final behavior of an object and execute
when the object is no longer in use. Although they are often used in C++ to free resources
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.

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

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

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

Output:

```
Constructed!
Finalized!
```
3.5 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:

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

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

An even shorter way for getter/setter methods are accessors that do both in one line:

```csharp
class Culture
{
    public int TalkedCountries { get; set; }
    public string Language { get; set; }
}

class InterculturalDialogue
{
    Culture culture;

    culture.Language = "Italian"; // ==> culture.SetLanguage("Italian");

    string strThisLanguage = culture.Language; // ==> ...
    culture.GetLanguage();
}
```

The code is equivalent to a GetLanguage and SetLanguage method definition, but without having to define these methods. The user can directly access the member, if it is not private, of course.

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.

```csharp
public class MyProgram
{
    MyClass myClass = new MyClass;

    Console.WriteLine(myClass.IntegerField); // Writes 3 to the
```
Indexers

command line.

myClass.IntegerField = 7; // Indirectly assigns 7 to the field
myClass.m_iField

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

3.6 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:

```csharp
public string this[string strKey]
{
    get { return coll[strKey]; }
    set { coll[strKey] = 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:

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

3.7 Events

C# events are class members that expose notifications to clients of the class. Events are only getting fired and never assigned.

```csharp
using System;

// Note: You need to know some about delegate, properties and methods
to understand this sample
namespace EventSample
{
    /// <summary>
    /// This delegate defines the signature of the appropriate method
    /// </summary>
    public delegate void ContractHandler(Employee sender);

    /// <summary>
    /// Employee class
    /// </summary>
    public class Employee
    {
        /// <summary>
        /// Field for the info whether or not the Employee is
        ```
engaged
    /// </summary>
    private bool m_bIsEngaged = false;
    /// </summary>
    /// Age of the employee
    /// </summary>
    private int m_iAge = -1;
    /// </summary>
    /// Name of the employee
    /// </summary>
    private String m_strName = null;

    /// </summary>
    /// *** The our event ***
    /// Is a collection of methods that will be called when it fires
    /// </summary>
    public event ContractHandler Engaged;

    /// </summary>
    /// Standard constructor
    /// </summary>
    public Employee()
    {
        // Here, we are adding a new method with appropriate signature (defined by delegate)
        // note: when a event not have any method and it was fired, it causes a exception!
        // for all effects when programming with events, assign one private method to event
        // or simply do a verification before fire it! -->
        if (event != null)
            this.Engaged += new ContractHandler(this.OnEngaged);
    }

    /// </summary>
    /// Event handler for the "engaged" event
    /// </summary>
    /// <param name="sender">
    /// Sender object
    /// </param>
    private void OnEngaged(Employee sender)
    {
        Console.WriteLine("private void OnEngaged was called!
this employee is engaged now!");
    }

    /// </summary>
    /// Accessor for the employee name
    /// </summary>
    public string Name
    {
        get
        {
            return m_strName;
        }

        set
        {
            m_strName = value;
        }
    }

    /// </summary>
    /// Accessor for the employee age
    /// </summary>
    public int Age
    {
get
{
    return m_iAge;
}

set
{
    m_iAge = value;
}
}

/// <summary>
/// Accessor for the information about Employee engagement
/// </summary>
public bool IsEngaged
{
    get
    {
        return m_bIsEngaged;
    }
    set
    {
        if (m_bIsEngaged == false && value == true)
        {
            // here we fires event (call all the methods that it have)
            // all times when IsEngaged is false and set to true;
            Engaged(this);
        }
        m_bIsEngaged = value;
    }
}

/// <summary>
/// Class for the entry point
/// </summary>
public class EntryPointClass
{
    static void Main(string[] a_strArgs)
    {
        Employee simpleEmployee = new Employee();
        simpleEmployee.Age = 18;
        simpleEmployee.Name = "Samanta Rock";
        // Here...
        // This is saying when the event fire, the method added to event are called too.
        // note that we cannot use =
        // is only += to add methods to event or -= do retire a event
        simpleEmployee.Engaged += new ContractHandler(SimpleEmployee_Engaged);
        // make attention here...
        // when I assign true to this property, the event Engaged will be called
        // when event is called, all method that it have, are called!
        simpleEmployee.IsEngaged = true;
        Console.ReadLine();
    }
}
See also here\(^8\) for details.

### 3.8 Operator overloading

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:

```csharp
public class Complex
{
    private double m_dReal, m_dImaginary;

    public double Real
    {
        get { return m_dReal; }
        set { m_dReal = value; }
    }

    public double Imaginary
    {
        get { return m_dImaginary; }
        set { m_dImaginary = value; }
    }

    // binary operator overloading
    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex() { Real = c1.Real + c2.Real, Imaginary = c1.Imaginary + c2.Imaginary; }
    }

    // unary operator overloading
    public static Complex operator -(Complex c)
    {
        return new Complex() { Real = -c.Real, Imaginary = -c.Imaginary; }
    }

    // cast operator overloading (both implicit and explicit)
    public static implicit operator double(Complex c)
    {
        // return the modulus - sqrt(x^2 + y^2)
        return Math.Sqrt(Math.Pow(c.Real, 2) + Math.Pow(c.Imaginary, 2));
    }
}
```

\(^8\) Chapter 4.11 on page 83
public static explicit operator string(Complex c)
{
    // we should be overloading the ToString() method, but this
    // is just a demonstration
    return c.Real.ToString() + " + " + c.Imaginary.ToString() + "i";
}

public class StaticDemo
{
    public static void Main()
    {
        Complex number1 = new Complex() { Real = 1, Imaginary = 2 };  
        Complex number2 = new Complex() { Real = 4, Imaginary = 10 };   
        Complex number3 = number1 + number2; // number3 now has Real
        number3 = -number3; // number3 now has Real = -5, Imaginary =  
        number3 = -number3; // number3 now has Real = -5, Imaginary =  
        double testNumber = number3; // testNumber will be set to the
        Console.WriteLine((string)number3); // This will print "-5 +  
        // The cast to string was needed because that was an explicit
        // cast operator.
    }
}

3.9 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 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 reference 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:

```csharp
struct Employee
{
    public int m_iAge;
    private string m_strName;

    public string Name
    {
        get { return m_strName; }
    }
}
```

9 Chapter 3.14.2 on page 67
Since C# 2.0, it is possible to have arrays\(^{10}\) inside structures, but only in unsafe contexts:

```csharp
struct data
{
    int header;
    fixed int values[10];
}
```

The array is accessed using pointer arithmetic. Values are treated arrayed values as if they were C-style arrays using indexing, etc.

### 3.9.1 Structure constructors

Structures need constructors - or better to say initialisers, as they do not construct but just initialise the memory\(^{11}\) - so that their contents are not left uninitialised. Therefore, constructors without parameters are not allowed.

Structure variables can be assigned one to another if and only if the structure variable on the right side of the assignment are all initialised.\(^{13}\)

```csharp
struct Timestamp
{
    private ushort m_usYear;
    private ushort m_usMonth;
    private ushort m_usDayOfMonth;
    private ushort m_usHour;
    private ushort m_usMinute;
    private ushort m_usSecond;

    public Timestamp(ushort usYear,
                     ushort usMonth,
                     ushort usDay,
                     ushort usHour,
                     ushort usMinute,
                     ushort usSecond)
    {
        m_usYear = usYear - 1900;
        m_usMonth = usMonth;
        m_usDay = usDay;
        m_usHour = usHour;
        m_usMinute = usMinute;
        m_usSecond = usSecond;
    }
}
```

---

10 Chapter 2.9.4 on page 21
11 Structure constructors \(^{12}\). MSDN. Retrieved 2012-04-12 [http://](http://)
3.10 Static classes

Static classes are commonly used to implement a Singleton Pattern\(^{15}\). 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:

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

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

3.11 References

16

3.12 Introduction

The .NET framework consists of several languages, all which follow the "object orientated programming" (OOP) approach to software development. This standard defines that all objects support

- Inheritance - the ability to inherit and extend existing functionality.
- Encapsulation - the ability to allow the user to only see specific parts, and to interact with it in specific ways.
- Polymorphism - the ability for an object to be assigned dynamically, but with some predictability as to what can be done with the object.

Objects are synonymous with objects in the real world. Think of any object and think of how it looks and how it is measured and interacted with. When creating OOP languages, the reasoning was that if it mimics the thought process of humans, it would simplify the coding experience.

For example, let’s consider a chair, and its dimensions, weight, colour and what is it made out of. In .NET, these values are called 'Properties'. These are values that define the


\(^{16}\) http://en.wikibooks.org/wiki/Category%20Sharp%20Programming
object's state. Be careful, as there are two ways to expose values: Fields and Properties. The recommended approach is expose Properties and not fields.

So we have a real-world idea of the concept of an object. In terms of practicality for a computer to pass information about, passing around an object within a program would consume a lot of resources. Think of a car, how many properties that has - 100's, 1000's. A computer passing this information about all the time will waste memory, processing time and therefore a bad idea to use. So objects come in two flavours:

- Reference types
- Value types

### 3.13 Reference and Value Types

A reference type is like a pointer to the value. Think of it like a piece of paper with a street address on it, and the address leads to your house - your object with hundreds of properties. If you want to find it, go to where the address says! This is exactly what happens inside the computer. The reference is stored as a number, corresponding to somewhere in memory where the object exists. So instead of moving an object around - like building a replica house every time you want to look at it - you just look at the original.

A value type is the exact value itself. Values are great for storing small amounts of information: numbers, dates etc.

There are differences in the way they are processed, so we will leave that until a little later in the article.

As well as querying values, we need a way to interact with the object so that some operation can be performed. Think of files - it's all well and good knowing the length of the file, but how about `Read()`ing it? Therefore, we can use something called methods as a way of performing actions on an object.

An example would be a rectangle. The properties of a rectangle are:

- Length
- Width

The 'functions' (or methods in .NET) would be:

- Area (\(= \text{Length} \times \text{Width}\))
- Perimeter (\(= 2\times\text{Length} + 2\times\text{Width}\))

Methods vary from Properties because they require some transformation of data to achieve a result. Methods can either return a result (such as Area) or not. Like above with the chair, if you `Sit()` on the chair, there is no expected reaction, the chair just ... works!

#### 3.13.1 System.Object

To support the first rule of OOP - Inheritance, we define something that all objects will derive from - this is `System.Object`, also known as `Object` or `object`. This object defines some methods that all objects can use should they need to. These methods include:
Object basics

- GetHashCode() - retrieve a number unique to that object.
- GetType() - retrieves information about the object like method names, the objects name etc.
- ToString() - convert the object to a textual representation - usually for outputting to the screen or file.

Since all objects derive from this class (whether you define it or not), any class will have these three methods ready to use. Since we always inherit from System.Object, or a class that itself inherits from System.Object, we therefore enhance and/or extend its functionality. Like in the real world that humans, cats, dogs, birds, fish are all an improved and specialised version of an 'organism'.

3.14 Object basics

All objects by default are reference types. To support value types, objects must instead inherit from the System.ValueType abstract class, rather than System.Object.

3.14.1 Constructors

When objects are created, they are initialized by the 'constructor'. The constructor sets up the object, ready for use. Because objects need to be created before being used, the constructor is created implicitly, unless it is defined differently by the developer. There are 3 types of constructor:

- Copy Constructor
- Static Constructor
- Default constructor - takes no parameters.
- Overloaded constructor - takes parameters.

Overloaded constructors automatically remove the implicit default constructor, so a developer must explicitly define the default constructor, if they want to use it.

A constructor is a special type of method in C# that allows an object to initialize itself when it is created. If a constructor method is used, there is no need to write a separate method to assign initial values to the data members of an object.

Important characteristics of a constructor method:

1. A constructor method has the same name as the class itself.
2. A constructor method is usually declared as public.
3. Constructor method is declared as public because it is used to create objects from outside the class in which it is declared. We can also declare a constructor method as private, but then such a constructor cannot be used to create objects.
4. Constructor methods do not have a return type (not even void).
5. C# provides a default constructor to every class. This default constructor initializes the data members to zero. But if we write our own constructor method, then the default constructor is not used.
6. A constructor method is used to assign initial values to the member variables.
7. The constructor is called by the new keyword when an object is created.
8. We can define more than one constructor in a class. This is known as constructor overloading. All the constructor methods have the same name, but their signatures are different, i.e., number and type of parameters are different.
9. If a constructor is declared, no default constructor is generated.

Copy Constructor

A copy constructor creates an object by copying variables from another object. The copy constructor is called by creating an object of the required type and passing it the object to be copied.

In the following example, we pass a Rectangle object to the Rectangle constructor so that the new object has the same values as the old object.

```csharp
using System;

namespace CopyConstructor
{

class Rectangle
{
    public int length;
    public int breadth;

    public Rectangle(int x, int y) // constructor fn
    {
        length = x;
        breadth = y;
    }

    public Rectangle(Rectangle r)
    {
        length = r.length;
        breadth = r.breadth;
    }

    public void display()
    {
        Console.WriteLine("Length = "+ length);
        Console.WriteLine("Breadth = "+ breadth);
    }
}

class Program
{
    public static void Main()
    {
        Rectangle r1 = new Rectangle(5, 10);
        Console.WriteLine("Values of first object");
        r1.display();

        Rectangle r2 = new Rectangle(r1);
        Console.WriteLine("Values of second object");
        r2.display();

        Console.ReadLine();
    }
}
}
Static Constructor

A static constructor is first called when the runtime first accesses the class. Static variables are accessible at all times, so the runtime must initialize it on its first access. The example below, when stepping through in a debugger, will show that `static MyClass()` is only accessed when the `MyClass.Number` variable is accessed.

C# supports two types of constructors: static constructor and instance constructor. Whereas an instance constructor is called every time an object of that class is created, the static constructor is called only once. A static constructor is called before any object of the class is created, and is usually used to initialize any static data members of a class.

A static constructor is declared by using the keyword static in the constructor definition. This constructor cannot have any parameters or access modifiers. In addition, a class can only have one static constructor. For example:

```csharp
using System;
using System.Collections.Generic;
using System.Text;

namespace StaticConstructors
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            int j = 0;
            Console.WriteLine("Static Number = " + MyClass.Number);
        }
    }

    class MyClass
    {
        private static int _number;
        public static int Number { get { return _number; } }
        static MyClass()
        {
            Random r = new Random();
            _number = r.Next();
        }
    }
}
```

Default Constructor

The default constructor takes no parameters and is implicitly defined, if no other constructors exist. The code sample below show the before, and after result of creating a class.

```csharp
// Created by the developer
class MyClass
{
}

// Created by the compiler
class MyClass : System.Object
{
    public MyClass() : base()
    {
```
Overloaded Constructors

To initialize objects in various forms, the constructors allow customization of the object by passing in parameters.

class MyClass
{
    private int _number;
    public int Number { get { return _number; } }

    public MyClass()
    {
        Random randomNumber = new Random();
        _number = randomNumber.Next();
    }

    public MyClass(int seed)
    {
        Random randomNumber = new Random(seed);  
        _number = randomNumber.Next();
    }
}

Calling other constructors

To minimise code, if another constructor implements the functionality better, you can instruct the constructor to call an overloaded (or default) constructor with specific parameters.

class MyClass
{
    private int _number;
    public int Number { get { return _number; } }

    public MyClass() : this ( DateTime.Now.Milliseconds ) // Call the other constructor passing in a value.
    {
    }

    public MyClass(int seed)
    {
        Random r = new Random(seed);  
        _number = r.Next();
    }
}

Base classes constructors can also be called instead of constructors in the current instance.

class MyException : Exception
{
    private int _number;
    public int Number { get { return _number; } }

    public MyException ( int errorNumber, string message, 
    Exception innerException)
Object basics

3.14.2 Destructors

As well as being "constructed", objects can also perform cleanup when they are cleared up by the garbage collector. As with constructors, the destructor also uses the same name as the class, but is preceded by the tilde (~) sign. However, the garbage collector only runs when either directly invoked, or has reason to reclaim memory, therefore the destructor may not get the chance to clean up resources for a long time. In this case, look into use of the Dispose() method, from the IDisposable interface.

Destructors are recognised via the use of the ~ symbol in front of a constructor with no access modifier. For example:

```csharp
class MyException : Exception
{
    private int _number;
    public int Number { get { return _number; } }

    public MyException (int errorNumber, string message, Exception innerException)
    : base(message, innerException)
    {
        _number = errorNumber;
    }

    ~MyException()
    {
    }
}
```

Encapsulation is depriving the user of a class of 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 that are not allowed to outside users. The private protection level is default for all class and struct members. This means that if you do not specify the protection modifier of a method or variable, it is considered as private by the compiler.

For example:

```csharp
public class Frog
{
    private int _height = 0;

    // Methods
    public void JumpLow() { Jump(1); }
    public void JumpHigh() { Jump(10); }
}
```
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 they 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.

3.15 Protection Levels

3.15.1 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. If no protection level is specified, class members will default to private.

```csharp
namespace PrivateSample
{
    public class Person
    {
        private string _name;

        // Methods
        public Person(string name)
        {
            this._name = name;
        }
    }
}
```

```csharp
public class Entry
{
    static void Main(string[] args)
    {
        Person OnePerson = new Person("Samanta");
        //OnePerson._name = "Sam"; // This causes a error of access level
    }
}
```

3.15.2 Protected

Protected members can be accessed by the class itself and by any class derived from that class.

```csharp
namespace ProtectedSample
```

17 Chapter 3.5 on page 54
{ public class Person
{
    protected string _name;
}
/// <summary>
/// When a class inherits from other class, it can access your
protected and public members
/// above your created members
/// </summary>
public class Warrior : Person
{
    public voidSetName(string name)
    {
        // Protected members can be accessed by internal methods
        // or constructors of class
        // so, it can be accessed by inherit class too
        base._name = name;
    }
}
}

public class Entry
{
    static void Main(string[] args)
    {
        Warrior OnePerson = new Warrior();
        OnePerson.SetName("Glades"); // OK
        // OnePerson._name = "Sam"; // This causes a error of
        access level too
        // protected members can not be accessed by external
        scopes
    }
}

3.15.3 Public

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

namespace PublicSample
{
    public class Person
    {
        public string Name;
    }

    public class Entry
    {
        static void Main(string[] args)
        {
            Person BeautifulPerson = new Person();
            BeautifulPerson.Name = "Debora"; // OK, public member can
            be accessed by other scopes
        }
    }
}

It is good programming practice not to expose member variables to the outside, unless it is
necessary. This is true especially for fields that should only be accessible over accessor\textsuperscript{18}

\textsuperscript{18} \url{http://en.wikibooks.org/wiki/%3AW%3AAccessor%20method}
and mutator method\(^{19}\)'s (getters and setters). Exceptions are member variables that are constant.

### 3.15.4 Internal

Internal members are accessible only in the same assembly and invisible outside it. If no protection level is specified for top level classes, they are treated as internal, and can only be accessed within the assembly.

```csharp
namespace InternalSample
{
    public class Person
    {
        internal string Name;
    }

    public class Entry
    {
        static void Main(string[] args)
        {
            Person BeautifulPerson = new Person();
            BeautifulPerson.Name = "Debora"; // OK, internal member can be accessed by other
            // scopes in same assembly supposing that Person is in another assembly, by example a
            // library, the name cannot be accessed. In another assembly source, this causes an error:
            // BeautifulPerson.Name = "Debora"; // Cannot access internal member
        }
    }
}
```

### 3.15.5 Protected Internal

Protected internal members are accessible from any class derived from the that class, or any class within the same assembly. So, it means protected or internal\(^{20}\).

Here, an example:

```csharp
namespace InternalSample
{
    public class Person
    {
        protected internal string Name;
    }

    public class Entry
    {
        static void Main(string[] args)
        {
            Person BeautifulPerson = new Person();
            BeautifulPerson.Name = "Debora"; // As above...
        }
    }
}
```


\(^{20}\) Type Member Access Modifiers 21]. C# STATION. Retrieved 2011-08-12 [http://](http://)
public class Book : InternalSample.Person
{
    static void Main(string[] args)
    {
        string aName = BeautifulPerson.Name; // Can be accessed, as Book is derived from Person
    }
}

3.16 References
4 Advanced Concepts

4.1 Inheritance

Inheritance is the ability to create a class from another class, the "parent" class, extending the functionality and state of the parent in the derived, or "child" class. It allows derived classes to overload methods from their parent class.

Inheritance is one of the pillars of object-orientation. It is the mechanism of designing one class from another and is one of the ideas for code reusability, supporting the concept of hierarchical classification. C# programs consist of classes, where new classes can either be created from scratch or by using some or all properties of an existing class.

Another feature related to inheritance and reusability of code is polymorphism, which permits the same method name to be used for different operations on different data types. Thus, C# supports code reusability by both features.

Important characteristics of inheritance include:

1. A derived class extends its base class. That is, it contains the methods and data of its parent class, and it can also contain its own data members and methods.
2. The derived class cannot change the definition of an inherited member.
3. Constructors and destructors are not inherited. All other members of the base class are inherited.
4. The accessibility of a member in the derived class depends upon its declared accessibility in the base class.
5. A derived class can override an inherited member.

An example of inheritance:

```csharp
using System;
using System.Text;

namespace ContainmentInheritance
{
    class Room
    {
        public int length;
        public int breadth;
        public int height;
        public string name;

        public Room(int l, int b, int h)
        {
            length = l;
            breadth = b;
            height = h;
        }
    }
}
```
class Home
{
    int numberOfRooms;
    int plotSize;
    string locality;

    // create an object of class Room inside class Home
    Room studyRoom = new Room(10, 12, 12);

    public Home()
    {
        numberOfRooms = 1;
        plotSize = 1000;
        locality = "Versova";
        name = "study room";
    }

    public void Display()
    {
        Console.WriteLine("MyHome has {0} rooms", numberOfRooms);
        Console.WriteLine("Plot size is {0}", plotSize);
        Console.WriteLine("Locality is {0}", locality);

        int area = studyRoom.length*studyRoom.breadth;
        Console.WriteLine("Area of the {0} room is {1}", name, area);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Home myhome = new Home();
        myhome.Display();
        Console.ReadLine();
    }
}

4.2 Subtyping Inheritance

The code sample below shows two classes, Employee and Executive. Employee has the 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.

public class Employee
{
    // We declare one method virtual so that the Executive class
    // can override it.
    public virtual void GetPayCheck()
    {
        // Get paycheck logic here.
    }
}
Now, we create an Executive class that will override the GetPayCheck method:

```csharp
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, as it is inherited from Employee.

```csharp
static void Main(string[] args)
{
    Employee emp = new Employee();
    Executive exec = new Executive();
    emp.Work();
    exec.Work();
    emp.GetPayCheck();
    exec.GetPayCheck();
    exec.AdministerEmployee();
}
```

### 4.3 Virtual Methods

If a base class contains a virtual method that it calls elsewhere and a derived class overrides that virtual method, the base class will actually call the derived class' method:

```csharp
public class Resource : IDisposable
{
    private bool _isClosed = false; // good programming practice initialise, although default

    protected virtual void Close()
    {
        Console.WriteLine("Base resource closer called!");
    }

    ~Resource()
    {
    }
}
```
Advanced Concepts

    Dispose();
    
    public void Dispose()
    {
        if (!_isClosed)
        {
            Console.WriteLine("Disposing resource and calling the
            Close() method...");
            _isClosed = true;
            Close();
        }
    }

public class AnotherTypeOfResource : Resource
{
    protected override void Close()
    {
        Console.WriteLine("Another type of resource closer called!");
    }
}

public class VirtualMethodDemo
{
    public static void Main()
    {
        Resource res = new Resource();
        AnotherTypeOfResource res2 = new AnotherTypeOfResource();
        res.Dispose(); // Resource.Close() will be called.
        res2.Dispose(); // Even though Dispose() is part of the
        Resource class, // the Resource class will call
        AnotherTypeOfResource.Close();
    }
}

4.4 Constructors

A derived class does not automatically inherit the base class' constructors, and it cannot
be instantiated unless it provides its own. A derived class must call one of its base class'
constructors by using the `base` keyword:

public class MyBaseClass
{
    public MyBaseClass(string text)
    {
        ...
    }
}

public class MyDerivedClass : MyBaseClass
{
    public MyDerivedClass(int number)
        : base(number.ToString())
    {
    }
    
    public MyDerivedClass(string text) // even though this is exactly
    the same as MyBaseClass,
    // only constructor, this is still necessary as constructors are
    not inherited.
        : base(text)
4.5 Inheritance keywords

The way C# inherits from another class syntactically is by using the : operator.

Example:

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

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

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

To override a method, use the `override` keyword:

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

A missing `new` or `override` keyword for a derived method may result in errors or warnings during compilation.:\(^1\) Here an example:

```csharp
abstract class ShapesA
{
    abstract public int Area(); // abstract!
}

class Square : ShapesA
{
    int x, y;

    public int Area() // Error: missing override, or ,new,
    {
        return x * y;
    }
}

class Shapes
{
    virtual public int Area() { return 0; } // it is virtual now!
}

class Square : Shapes
{
    int x, y;

    public int Area() // no explicit override, or ,new, required
    {
        return x * y;
    }
}
```

---

\(^1\) C# design: Why is new/override required on abstract methods but not on virtual methods? / Answer

The `Square` class method `Area()` will result in a compilation error, if it is derived from the `ShapesA` class:

```
```

The same method will result in a compilation warning, if derived from the normal `Shapes` class:

```
To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.
```

### 4.6 References

An INTERFACE in C# is a type definition similar to a class, except that it purely represents a contract between an object and its user. It can neither be directly instantiated as an object, nor can data members be defined. So, an interface is nothing but a collection of method and property declarations. The following defines a simple interface:

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

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 as `Draw()` in the above example. The interface name is then formed as an adjective by adding the "...able" suffix. So, the interface name above could also be `IDrawable`. This convention is used throughout the .NET Framework.

Implementing an interface is simply done by inheriting off it and defining all the methods and properties declared by the interface after that. For instance,

```csharp
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 inherit from one class only, it can inherit from any number of interfaces. This is a 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 instance, using the previous examples,

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

Interfaces 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();
}

4.7 Additional details

Access specifiers (i.e. private, internal, etc.) cannot be provided for interface members, as 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 child class.

There are no static methods within an interface, but any static methods can be implemented in a class that manages objects using it.

In addition to methods and properties, interfaces can declare events and indexers as well. For those familiar with Java, C#’s interfaces are extremely similar to Java’s.
4.8 Introduction

Delegates and events are fundamental to any Windows or Web Application, allowing the developer to "subscribe" to particular actions carried out by the user. Therefore, instead of expecting everything and filtering out what you want, you choose what you want to be notified of and react to that action.

A delegate is a way of telling C# which method to call when an event is triggered. For example, if you click a Button on a form, the program would call a specific method. It is this pointer that is a delegate. Delegates are good, as you can notify several methods that an event has occurred, if you wish so.

An event is a notification by the .NET framework that an action has occurred. Each event contains information about the specific event, e.g., a mouse click would say which mouse button was clicked where on the form.

Let's say you write a program reacting only to a Button click. Here is the sequence of events that occurs:

- User presses the mouse button down over a button
  - The .NET framework raises a MouseDown event
- User releases the mouse button
  - The .NET framework raises a MouseUp event
  - The .NET framework raises a MouseClick event
  - The .NET framework raises a Clicked event on the Button

Since the button's click event has been subscribed, the rest of the events are ignored by the program and your delegate tells the .NET framework which method to call, now that the event has been raised.

4.9 Delegates

Delegates form the basis of event handling in C#. They are a construct for abstracting and creating objects that reference methods and can be used to call those methods. 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:

```csharp
using System;
delegate void Procedure();

class DelegateDemo
{
    public static void Method1()
    {
```

delegate void Procedure();

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, but merely declares a delegate type called Procedure that takes no arguments and returns nothing. Next, in the Main() method, the statement Procedure someProcs = null; instantiates a delegate. The assignment means that the delegate is not initially referencing any methods. The statements someProcs += new Procedure(DelegateDemo.Method1) and someProcs += new Procedure(Method2); // Example with omitted class name add two static methods to the delegate instance. Note that the class name can also be left off, as the statement is occurring inside DelegateDemo. 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:

someProcs -= new Procedure(DelegateDemo.Method1);

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

someProcs += DelegateDemo.Method1;
someProcs -= 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, 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).

### 4.10 Anonymous delegates

Anonymous delegates are a short way to write delegate code, specified using the `delegate` keyword. The delegate code can also reference local variables of the function in which they are declared. Anonymous delegates are automatically converted into methods by the compiler. For example:

```csharp
using System;
delegate void Procedure();

class DelegateDemo2
{
    static Procedure someProcs = null;

    private static void AddProc()
    {
        int variable = 100;

        someProcs += new Procedure(delegate
        {
            Console.WriteLine(variable);
        });
    }

    static void Main()
    {
        someProcs += new Procedure(delegate {
            Console.WriteLine("test"); });
        AddProc();
        someProcs();
        Console.ReadKey();
    }
}
```

They can accept arguments just as normal methods can:

```csharp
using System;
delegate void Procedure(string text);

class DelegateDemo3
{
    static Procedure someProcs = null;

    private static void AddProc()
    {
        int variable = 100;

        someProcs += new Procedure(delegate(string text)
        {
            Console.WriteLine(text + ", " + variable.ToString());
        });
    }

    static void Main()
    {
```
someProcs += new Procedure(delegate(string text) {
    Console.WriteLine(text); });
AddProc();
someProcs("testing");
Console.ReadKey();
}
}
The output is:

```
testing
testing, 100
```
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 it.

Delegates and Events

In general terms, an interface is the set of public members of a component. Of course, this
is also true for C# interface. A C# class also defines an interface, as it has a set of public
members. A non-abstract C# class defines the implementation of each member.

In C#, it is possible to have a type that is intermediate between a pure interface that does
not define any implementation, and a type that defines a complete implementation. This
is called an abstract class and is defined by including the abstract keyword in the class
definition.

An abstract class is somewhere between a C# interface and a non-abstract class. Of the
public members defined by an abstract class, any number of those members may include an
implementation.

For example, an abstract class might provide an implementation for none of its members.

public abstract class AbstractShape
{
    public abstract void Draw(Graphics g);
    public abstract double X
}

4.12 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

5 Chapter 4.6 on page 78
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*)

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

    public bool Create()
    {
    }
}
```

**Listing 2**: Class split across multiple files

(*file1.cs*)

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

(*file2.cs*)

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

Generics are a new feature available since version 2.0 of the C# language and the common language runtime (CLR). Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. The most common use of generics is to create collection classes. Generic types were introduced to **maximize code reuse, type safety, and performance**.\(^7\)

---


\(^7\) Generics (C# Programming Guide) \(^8\). msdn . Retrieved 2011-08-09
4.13 Generic classes

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

```csharp
public class SomeObjectContainer
{
    private object _obj;

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

    public object GetObject()
    {
        return this._obj;
    }
}
```

And its usage would be:

```csharp
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
    }
}
```

Note that we have to cast back to original data type we have chosen (in this case - `int`) every time we want to get an object from such a container. In such small programs like this, everything is clear. But in more complicated cases with more containers in different parts of the program, we would have to take care that the container is supposed to have `int` type in it and no other data type, as in such a case, a `InvalidCastException` is thrown.

Additionally, if the original data type we have chosen is a value 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 a `try - catch` block, or we could create a 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 a 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 immediately after the 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>`.

```csharp
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:

```csharp
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 the type for a "container" once, avoiding previously mentioned problems and autoboxing for `structs`.

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

- You need to keep objects of a single type in a class
- You do not 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.

### 4.14 Generic interfaces

A generic interface accepts one or more type parameters, similar to a generic class:

```csharp
public interface IContainer<T>
{
}
```
Advanced Concepts

```csharp
T GetObject();
void SetObject(T value);
}

class Program
{
    static void Main(string[] args)
    {
        IContainer<string> container = new StringContainer();
        container.SetObject("test");
        Console.WriteLine(container.GetObject());
        container = new FileWithString();
        container.SetObject("another test");
        Console.WriteLine(container.GetObject());
        Console.ReadKey();
    }
}
```

Generic interfaces are useful when multiple implementations of a particular class are possible. For example, both the `List<T>` class (discussed below) and the `LinkedList<T>` class, both from the `System.Collections.Generic` namespace, implement the `IEnumerable<T>` interface. `List<T>` has a constructor that creates a new list based on an existing object that implements `IEnumerable<T>`, and so we can write the following:

```csharp
LinkedList<int> linkedList = new LinkedList<int>();
linkedList.AddLast(1);
linkedList.AddLast(2);
linkedList.AddLast(3);
// linkedList now contains 1, 2 and 3.
List<int> list = new List<int>(linkedList);
// now list contains 1, 2 and 3 as well!
```

### 4.15 Generic methods

Generic methods are very similar to generic classes and interfaces:
using System;
using System.Collections.Generic;

public static bool ArrayContains<T>(T[] array, T element)
{
    foreach (T e in array)
    {
        if (e.Equals(element))
        {
            return true;
        }
    }
    return false;
}

This method can be used to search any type of array:

using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        string[] strArray = { "string one", "string two", "string three" };  
        int[] intArray = { 123, 456, 789 };  

        Console.WriteLine(ArrayContains<string>(strArray, "string one"); // True
        Console.WriteLine(ArrayContains<int>(intArray, 135)); // False
    }
}

4.16 Type constraints

One may specify one or more type constraints in any generic class, interface or method using the where keyword. The following example shows all of the possible type constraints:

public class MyClass<T, U, V, W>
where T : class, // T should be a reference type (array, class, delegate, interface)
    new() // T should have a public constructor
with no parameters
    where U : struct // U should be a value type (byte, double, float, int, long, struct, uint, etc.)
        where V : MyOtherClass, // V should be derived from MyOtherClass
            IEnumerable<U> // V should implement IEnumerable<U>
        where W : T, // W should be derived from T
            IDisposable // W should implement IDisposable
    {
    ...
    }

These type constraints are often necessary to

1. create a new instance of a generic type (the new() constraint
2. use foreach on a variable of a generic type (the IEnumerable<T> constraint)
3. use `using` on a variable of a generic type (the `IDisposable` constraint)

### 4.17 Notes

Extension methods are a feature new to C# 3.0 and allow you to extend existing types with your own methods. While they are static, they are used as if they are normal methods of the class being extended. Thus, new functionality can be added to an existing class without a need to change or recompile the class itself. However, since they are not directly part of the class, extensions cannot access private or protected methods, properties, or fields.

Extension methods should be created inside a `static` class. They themselves should be `static` and should contain at least one parameter, the first preceded by the `this` keyword:

```csharp
public static class MyExtensions
{
    public static string[] ToStringArray<T>(this List<T> list)
    {
        string[] array = new string[list.Count];
        for (int i = 0; i < list.Count; i++)
            array[i] = list[i].ToString();
        return array;
    }
    // to be continued...
}
```

The type of the first parameter (in this case `List<T>`) specifies the type with which the extension method will be available. You can now call the extension method like this:

```csharp
List<int> list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
string[] strArray = list.ToStringArray(); // strArray will now contain "1", "2" and "3".
```

Here is the rest of the program:

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

public static class MyExtensions
{
    ... // continued from above
    public static void WriteToConsole(this string str)
    {
        Console.WriteLine(str);
    }
    public static string Repeat(this string str, int times)
    {
        return sb.ToString();
    }
}
```
class ExtensionMethodsDemo
{
    static void Main()
    {
        List<int> myList = new List<int>();
        for (int i = 1; i <= 10; i++)
            myList.Add(i);

        string[] myStringArray = myList.ToStringArray();
        foreach (string s in myStringArray)
            s.Repeat(4).WriteToConsole(); // string is extended by
            WriteToConsole();
        Console.ReadKey();
    }
}

Note that extension methods can take parameters simply by defining more than one parameter without the this keyword.

4.18 Introduction

All computer programs use up memory, whether that is a variable in memory, opening a file or connecting to a database. The question is how can the runtime environment reclaim any memory when it is not being used? There are three answers to this question:

- If you are using a managed resource, this is automatically released by the Garbage Collector
- If you are using an unmanaged resource, you must use the IDisposable interface to assist with the cleanup
- If you are calling the Garbage Collector directly, by using System.GC.Collect() method, it will be forced to tidy up resources immediately.

Before discussing managed and unmanaged resources, it would be interesting to know what the garbage collector actually does.

4.18.1 Garbage Collector

The garbage collector is a background process running within your program. It is always present within all .NET applications. Its job is to look for objects (i.e. reference types) which are no longer being used by your program. If the object is assigned to null, or the object goes out of scope, the garbage collector will mark the object be cleaned up at some point in the future, and not necessarily have its resources released immediately!
Why? The garbage collector will have a hard time keeping up with every de-allocation you make, especially at the speed the program runs and therefore only runs when resources become limited. Therefore, the garbage collector has three 'generations'.

- Generation 0 - the most recently created objects
- Generation 1 - the mid-life objects
- Generation 2 - the long term objects.

All reference types will exist in one of these three generations. They will firstly be allocated to Gen 0, then moved to Gen 1 and Gen 2 depending on their lifetime. The garbage collector works by removing only what is needed and so will only scan Gen 0 for a quick-fix solution. This is because most, if not all, local variables are placed in this area.

For more in-depth information, visit the MSDN Article\(^\text{11}\) for a better explanation.

Now you know about the garbage collector, let's discuss the resources that it is managing.

### 4.18.2 Managed Resources

Managed resources are objects which run totally within the .NET framework. All memory is reclaimed for you automatically, all resources closed and you are in most cases guaranteed to have all the memory released after the application closes, or when the garbage collector runs.

You do not have to do anything with them with regards to closing connections or anything, it is a self-tidying object.

### 4.18.3 Unmanaged Resources

There are circumstances where the .NET framework world will not release resources. This may be because the object references resources outside of the .NET framework, like the operating system, or internally references another unmanaged component, or that the resources accesses a component that uses COM, COM+ or DCOM.

Whatever the reason, if you are using an object that implements the `IDisposable` interface at a class level, then you too need to implement the `IDisposable` interface too.

```csharp
public interface IDisposable
{
    void Dispose();
}
```

This interface exposes a method called `Dispose()`. This alone will *not* help tidy up resources, as it is only an interface, so the developer must use it correctly in order to ensure the resources are released. The two steps are:

1. Always call `Dispose()` on any object that implements `IDisposable` as soon as you are finished using it. (This can be made easier with the `using` keyword)

2. Use the finalizer method to call Dispose(), so that if anyone has not closed your resources, your code will do it for them.

Dispose pattern

Often, what you want to clean up varies depending on whether your object is being finalized. For example, you would not want to clean up managed resources in a finalizer since the managed resources could have been reclaimed by the garbage collector already. The dispose pattern can help you implement resource management properly in this situation:

```csharp
public class MyResource : IDisposable
{
    private IntPtr _someUnmanagedResource;
    private List<long> _someManagedResource = new List<long>();

    public MyResource()
    {
        _someUnmanagedResource = AllocateSomeMemory();
        for (long i = 0; i < 10000000; i++)
            _someManagedResource.Add(i);
        ...
    }

    // The finalizer will call the internal dispose method, telling it not to free managed resources.
    ~MyResource()
    {
        this.Dispose(false);
    }

    // The internal dispose method.
    private void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Clean up managed resources
            _someManagedResource.Clear();
        }

        // Clean up unmanaged resources
        FreeSomeMemory(_someUnmanagedResource);
    }

    // The public dispose method will call the internal dispose method, telling it to free managed resources.
    public void Dispose()
    {
        this.Dispose(true);
        // Tell the garbage collector to not call the finalizer because we have already freed resources.
        GC.SuppressFinalize(this);
    }
}
```
4.18.4 Applications

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

```vbnet
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:

```csharp
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:

```il
.method public hidebysig static string Read(string FileName) cil
    .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)
    // 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)
```

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\textsuperscript{13}.

See the following sections for more applications of this technique.

### 4.18.5 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\textsuperscript{14}. Wikipedia has a brief note on the subject as well: Resource Acquisition Is Initialization\textsuperscript{15}.

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).

\textsuperscript{13} http://www.codeproject.com/csharp/tinguusingstatement.asp  
\textsuperscript{14} http://www.hackcraft.net/raii/  
\textsuperscript{15} http://en.wikipedia.org/wiki/Resource%20Acquisition%20Is%20Initialization  
\textsuperscript{16} http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming
Design Patterns are common building blocks designed to solve everyday software issues. Some basic terms and example of such patterns include what we see in everyday life. Key patterns are the singleton pattern, the factory pattern, and chain of responsibility patterns.

### 4.19 Factory Pattern

The factory pattern is a method call that uses abstract classes and its implementations, to give the developer the most appropriate class for the job.

Let's create a couple of classes first to demonstrate how this can be used. Here we take the example of a bank system.

```csharp
public abstract class Transaction
{
    private string _sourceAccount;

    // May not be needed in most cases, but may on transfers, closures and corrections.
    private string _destinationAccount;

    private decimal _amount;
    public decimal Amount { get { return _amount; } }

    private DateTime _transactionDate;
    private DateTime _effectiveDate;

    public Transaction(string source, string destination, decimal amount)
    {
        _sourceAccount = source;
        _destinationAccount = destination;
        _amount = amount;
        _transactionDate = DateTime.Now;
    }

    public Transaction(string source, string destination, decimal amount, DateTime effectiveDate) : this(source, destination, amount)
    {
        _effectiveDate = effectiveDate;
    }

    protected decimal AdjustBalance(string accountNumber, decimal amount)
    {
        decimal newBalance = decimal.MinValue;
        using(Mainframe.ICOMInterface mf = new Mainframe.COMInterfaceClass())
        {
            string dateFormat = DateTime.Now.ToString("yyyyMMdd HH:mm:ss");
            mf.Credit(dateFormat, accountNumber, amount);
            newBalance = mf.GetBalance(DateTime.Now.AddSeconds(1), accountNumber);
        }
        return newBalance;
    }

    public abstract bool Complete();
}
```
This Transaction class is incomplete, as there are many types of transactions:

- Opening
- Credits
- Withdrawals
- Transfers
- Penalty
- Correction
- Closure

For this example, we will take credit and withdrawal portions, and create classes for them.

```csharp
public class Credit : Transaction
{
    // Implementations hidden for simplicity
    public override bool Complete()
    {
        this.AdjustBalance(_sourceAccount, amount);
    }
}

public class Withdrawal : Transaction
{
    // Implementations hidden for simplicity
    public override bool Complete()
    {
        this.AdjustBalance(_sourceAccount, -amount);
    }
}
```

The problem is that these classes do much of the same thing, so it would be helpful, if we could just give it the values, and it will work out what class type we require. Therefore, we could come up with some ways to distinguish between the different types of transactions:

- Positive values indicate a credit.
- Negative values indicate a withdrawal.
- Having two account numbers and a positive value would indicate a transfer.
- Having two account numbers and a negative value would indicate a closure.
- etc.

So, let us write a new class with a static method that will do this logic for us, ending the name `Factory`:

```csharp
public class TransactionFactory
{
    public static Transaction Create(string source, string destination, decimal amount)
    {
        if (string.IsNullOrEmpty(destination))
        {
            if (amount >= 0)
                return new Credit(source, null, amount);
            else
                return new Withdrawal(source, null, amount);
        }
        else
        {
            // Other implementations here
        }
    }
}
```
Now, you can use this class to do all of the logic and processing, and be assured that the type you are returned is correct.

```csharp
public class MyProgram
{
    static void Main()
    {
        decimal randomAmount = new Random().Next() * 1000000;
        Transaction t = TransactionFactory.Create("123456", ",", randomAmount);
        // t.Complete(); <-- This would carry out the requested transaction.

        Console.WriteLine("(0): {1:C}, t.GetType().Name, t.Amount");
    }
}
```

### 4.20 Singleton

The singleton pattern instantiates only 1 object, and reuses this object for the entire lifetime of the process. This is useful, if you wish the object to maintain state, or if it takes lots of resources to set the object up. Below is a basic implementation:

```csharp
public class MySingletonExample
{
    private static Hashtable sharedHt = new Hashtable();

    public Hashtable Singleton
    {
        get
        {
            return sharedHt;
        }
        // set { ; }
        // Not implemented for a true singleton
    }

    // Class implementation here..
}
```

The **Singleton** property will expose the same instance to all callers. Upon the first call, the object is initialised and on subsequent calls this is used.

Examples of this pattern include:

- **ConfigurationSettings** (Generic settings reader)
- **HttpApplication** (Application object in ASP.NET)
- **HttpCacheUtility** (Cache object in ASP.NET)
- **HttpServerUtility** (Server object in ASP.NET)

17 http://en.wikibooks.org/wiki/Category%3A
5 The .NET Framework

.NET Framework is a common environment for building, deploying, and running Web Services, Web Applications, Windows Services and Windows 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.

5.1 Introduction

In June 2000 Microsoft released both the .NET platform and a new program language called C#. C# is a general-purpose OOP language designed to give optimum simplicity, expansiveness, and performance. Its syntax is very similar to Java, with the major difference being that all variable types are derived from a common ancestor class.

C# is a language in itself. It can perform mathematical and logical operations, variable assignment and other expected traits of a programming language. This in itself is not flexible enough for more complex applications. At some stage, the developer will want to interact with the host system whether it be reading files or downloading content from the Internet.

The .NET framework is a toolset developed for the Windows platform to allow the developer to interact with the host system or any external entity whether it be another process, or another computer. The .NET platform is a Windows platform-specific implementation. Other operating systems have their own implementations due to the differences in the operating systems I/O management, security models and interfaces.

5.2 Background

- Originally called NGWS (Next Generation Windows Services).
- .NET does not run in any browser. It is a runtime language (Common Language Runtime) like the Java runtime. By contrast, Microsoft Silverlight does run in a browser.
- .NET is based on the newest Web standards.
- .NET is built on the following Internet standards:
  - HTTP, the communication protocol between Internet Applications
  - SOAP, the standard format for requesting Web Services
  - UDDI, the standard to search and discover Web Services
  - XML, the format for exchanging data between Internet Applications

1 http://en.wikibooks.org/wiki/%3A%20Language%20Runtime
2 http://en.wikipedia.org/wiki/Microsoft%20Silverlight
5.3 Console Programming

5.3.1 Input

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

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

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.

5.3.2 Output

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

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

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 `\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).

### 5.3.3 Error

The `Error` output is used to divert error specific messages to the console. To a novice user this may seem fairly pointless, as this achieves the same as Output\(^4\) (as above). If you decide to write an application that runs another application (for example a scheduler), you may wish to monitor the output of that program - more specifically, you may only wish to be notified only of the errors that occur. If you coded your program to write to the `Console.Error` stream whenever an error occurred, you can tell your scheduler program to monitor this stream, and feedback any information that is sent to it. Instead of the Console appearing with the Error messages, your program may wish to log these to a file.

You may wish to revisit this after studying Streams and after learning about the Process class.

### 5.3.4 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.

c sharp is object based programming language. .net framework is a Microsoft programming language is used to create web application, console application, mobile application.

```csharp
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 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':

---

4 Chapter 5.3.2 on page 100
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 can check to see if the user entered all the required arguments.

```csharp
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 `args.Length` property returns the total number of arguments. If no arguments are given, it will return zero.

You are also able to group a single argument together by using the quote marks (" "). This is particularly useful if you are expecting many parameters, but there is a requirement for including spaces (e.g. file locations, file names, full names etc.)

```csharp
using System;
class Test
{
    public static void Main(string[] args)
    {
        for (int index = 0; index < args.Length; index++)
        {
            Console.WriteLine((index + 1) + ": " + args[index]);
        }
    }
}
```

5.3.5 Formatted output

`Console.Write()` and `Console.WriteLine()` allow you to output a text string, but also allows writing a string with variable substitution.

These two functions normally have a string as the first parameter. When additional objects are added, either as parameters or as an array, the function will scan the string to substitute objects in place of tokens.
For example:

```csharp
{ int i = 10;
  Console.WriteLine("i = \{0\}", i);
}
```

The `{0}` is identified by braces, and refers to the parameter index that needs to be substituted. You may also find a format specifier within the braces, which is preceded by a colon and the specifier in question (e.g. `{0:C}`).

### 5.3.6 Rounding number example

This is a small example that rounds a number to a string. It is an augmentation for the `Math` class of C#. The result of the Round method has to be rounded to a string, as significant figures may be trailing zeros that would disappear, if a number format would be used. Here is the code and its call. You are invited to write a shorter version that gives the same result, or to correct errors!

The constant class contains repeating constants that should exist only once in the code so that to avoid inadvertant changes. (If the one constant is changed inadvertantly, it is most likely to be seen, as it used at several locations.)

```csharp
using System;
using System.Globalization;
using System.IO;
using System.Text;
namespace ConsoleApplicationCommons
{
/// <summary>
/// Class for special mathematical calculations.
/// </summary>
using System;
using System.Globalization;
using System.IO;
using System.Text;
namespace ConsoleApplicationCommons
{
/// <summary>
/// Class for special mathematical calculations.
```
/// ATTENTION: Should not depend on any other class except Java libraries!
/// </summary>
public class Maths
{
    public static CultureInfo invC = CultureInfo.InvariantCulture;
    /// Value after which the language switches from scientific to double
    /// </summary>
    private const double E_TO_DOUBLE = 1E-4;
    /// Maximal digits after which Convert.ToString(...) becomes inaccurate.
    /// </summary>
    private const short MAX_CHARACTERS = 16;
    private static String strZeros = "000000000000000000000000000000000";

    /// Determines language-independently whether or not the character can be a decimal separator or not
    /// </summary>
    /// <param name="character">Character to be checked</param>
    /// <returns>true, if it can be a decimal separator in a language, and false otherwise.</returns>
    private static bool IsDecimalSeparator(char c)
    {
        return ((c == Common.COMMA) || (c == Common.PERIOD));
    }

    /// Determines how many zeros are to be appended after the decimal digits.
    /// </summary>
    /// <param name="separator">Language-specific decimal separator</param>
    /// <param name="d">Rounded number</param>
    /// <param name="significantsAfter">Significant digits after decimal</param>
    /// <returns>Requested value</returns>
    private static short CalculateMissingSignificantZeros(char separator, double d, short significantsAfter)
    {
        short after = FindSignificantsAfterDecimal(separator, d);
        short zeros = (short)(significantsAfter - ((after == 0) ? 1 : after));
        return (short)((zeros >= 0) ? zeros : 0);
    }

    /// Finds the decimal position language-independently.
    /// </summary>
    /// <param name="value">Value to be searched for the decimal separator

104
/// <param name="separator">Language-specific decimal separator</param>
/// <returns>Number of insignificant zeros after decimal separator.</returns>
private static short FindSignificantsAfterDecimal(char separator, double d)
{
    // Find the position of the first non-zero digit:
    short nonZeroAt = 0;
    for (; (nonZeroAt < mantissa.Length) && (mantissa[nonZeroAt] == '0'); nonZeroAt++)
    {
        mantissa = mantissa.Substring(nonZeroAt).Length;
    }
}

/// <summary>
/// Finds the significant digits after the decimal separator of a mantissa.
/// </summary>
/// <param name="separator">Language-specific decimal separator</param>
/// <returns>Number of insignificant zeros after decimal separator.</returns>
private static short FindSignificantsAfterDecimal(char separator, double d)
if (d == 0) return 1;
else
{
    String value = ConvertToString(d);

    short separatorAt =
    FindDecimalSeparatorPosition(value);

    if (separatorAt == -1) value =
    value.Substring(separatorAt + 1);

    short eAt = (short) value.IndexOf(Common.EXPONENT);
    if ((separatorAt == -1) && (eAt == -1)) return 0;
    else if (eAt > 0) value = value.Substring(0, eAt);

    long longValue = Convert.ToInt64(value, invC);
    if (longValue == 0) return 0;
    else if (Math.Abs(d) < 1)
    {
        value = Convert.ToString(longValue, invC);
        if (value.Length >= 15)
        {
            return (byte)Convert.ToString(longValue,
            invC).Length;
        }
        else return (byte)(value.Length);
    }
    else
    {
        if (value.Length >= 15) return
        (byte)(value.Length - 1);
        else return (byte)(value.Length);
    }
}

/// <summary>
/// Determines the number of significant digits after the
decimal
/// separator knowing the total number of significant digits
/// and
/// the number before the decimal separator.
/// </summary>
/// <param name="significantsBefore">Number of significant digits before separator</param>
/// <param name="significantDigits">Number of all significant digits</param>
/// <returns>
/// Number of significant decimals after the separator
/// </returns>
private static short FindSignificantsAfterDecimal(
    short significantsBefore,
    short significantDigits)
{
    short significantsAfter =
    (short)(significantDigits - significantsBefore);

    return (short)((significantsAfter > 0) ?
    significantsAfter : 0);
}

/// <summary>
/// Determines the number of digits before the decimal point.
/// </summary>
/// <summary>
/// Returns the exponent part of the double number.
/// </summary>
/// <param name="d">Value of which the exponent is of interest</param>
/// <returns>Exponent of the number or zero.</returns>
private static short FindExponent(double d)
{
    return short.Parse(FindExponent(Convert.ToString(d, invC)), invC);
}

/// <summary>
/// Finds the exponent of a number.
/// </summary>
/// <param name="value">Value where an exponent is to be searched</param>
/// <returns>Exponent, if it exists, or "0".</returns>
private static string FindExponent(string value)
{
    short eAt = (short)(value.IndexOf(Common.EXPONENT));
    if (eAt < 0) return Common.ZERO;
    else
    {
        return Convert.ToString(short.Parse(value.Substring(eAt + 1)), invC);
    }
}
/// <summary>
/// Finds the mantissa of a number.
/// </summary>
/// <param name="separator">
/// Language-specific decimal separator
/// </param>
/// <param name="value">Value where the mantissa is to be found</param>
/// <returns>Mantissa of the number</returns>
private static String FindMantissa(char separator, String value)
{
    short eAt = (short)(value.IndexOf(Common.EXPONENT));
    if (eAt > -1) value = value.Substring(0, eAt);
    if (FindDecimalSeparatorPosition(value) == -1) value += ".0";
    return value;
}

/// <summary>
/// Retrieves the digits of the value only
/// </summary>
/// <param name="d">Number</param>
/// <returns>The digits only</returns>
private static String RetrieveDigits(double d)
{
    double dValue = d;
    short exponent = FindExponent(d);
    StringBuilder value = new StringBuilder();
    if (exponent == 0)
    {
        value.Append(dValue);
        if (value.Length >= MAX_CHARACTERS)
        {
            value.Clear();
            if (Math.Abs(dValue) < 1) value.Append("0");
            // Determine the exponent for a scientific form:
            exponent = 0;
            while (((long)dValue != dValue) && (dValue < 1E11))
            {
                dValue *= 10;
                exponent++;
            }
            value.Append((long)dValue);
            while ((long)dValue != dValue)
            {
                dValue -= (long)dValue;
                dValue *= 10;
                value.Append((long)dValue);
            }
        }
    } else 
    {
        
    }
double multiplier = Math.Pow(10, -exponent);
for (short s = 0; (s <= 16) && (exponent != 0); s++)
{
    dValue *= multiplier;
    value.Append((long)dValue);
    dValue -= (long)dValue;
    exponent++;
    multiplier = 10;
}

if (value.Length >= MAX_CHARACTERS + 2)
    value.Length = MAX_CHARACTERS + 2;
return RetrieveDigits(value.ToString());

/// <summary>
/// Retrieves the digits of the value only
/// </summary>
/// <param name="number">Value to be scrutinised</param>
/// <returns>The digits only</returns>
private static String RetrieveDigits(String number)
{
    // Strip off exponent part, if it exists:
    short eAt = (short)number.IndexOf(Common.EXPONENT);
    if (eAt > -1) number = number.Substring(0, eAt);
    return number.Replace(Convert.ToString(Common.DASH), "").Replace(  
        Convert.ToString(Common.COMMA), ",").Replace(  
        Convert.ToString(Common.PERIOD), "\n");
}

/// <summary>
/// Inserts the decimal separator at the right place
/// </summary>
/// <param name="dValue">Number</param>
/// <param name="value">String variable, where the separator is to be added.</param>
private static void InsertSeparator(double dValue, StringBuilder value)
{
    short separatorAt = (short)Convert.ToString((long)dValue).Length;
    if (separatorAt < value.Length)
        value.Insert(separatorAt, Common.PERIOD);
}

/// <summary>
/// Calculates the power of the base to the exponent without
/// changing
/// the least-significant digits of a number.
/// </summary>
/// <param name="basis"></param>
/// <param name="exponent">basis to power of exponent</param>
/// <returns></returns>
public static double Power(int basis, short exponent)
{
    return Power((short)basis, exponent);
}
/// Calculates the power of the base to the exponent without changing
/// the least-significant digits of a number.
/// </summary>
/// <param name="basis"> </param>
/// <param name="exponent"> </param>
/// <returns>basis to power of exponent</returns>
public static double Power(short basis, short exponent)
{
    if (basis == 0) return (exponent != 0) ? 1 : 0;
    else
    {
        if (exponent == 0) return 1;
        else
        {
            // The Math method power does change the least
            significant
            // digits after the decimal separator and is therefore
            // useless.
            long result = 1;
            short s = 0;
            if (exponent > 0)
            {
                for (; s < exponent; s++) result *= basis;
            }
            else if (exponent < 0)
            {
                for (s = exponent; s < 0; s++) result /=
            }
            return result;
        }
    }
}

/// <summary>
/// Rounds a number to the decimal places.
/// </summary>
/// <param name="d">Number to be rounded</param>
/// <param name="separator">Language-specific decimal separator</param>
/// <param name="significantsAfter">Number of decimal places after the separator</param>
/// <returns>Rounded number to the requested decimal places</returns>
public static double Round(char separator, double d, short significantsAfter)
{
    if (d == 0) return 0;
    else
    {
        double constant = Power(10, significantsAfter);
        short dsExponent = FindExponent(d);
        short exponent = dsExponent;

        double value = d*constant=Math.Pow(10, -exponent);
        String exponentSign = (exponent < 0) ? Convert.ToString(Common.DASH) : "";

        if (exponent != 0)
        {
            // The Math method power does change the least
            significant
            // digits after the decimal separator and is therefore
            // useless.
            long result = 1;
            short s = 0;
            if (exponent > 0)
            {
                for (; s < exponent; s++) result *= basis;
            }
            else if (exponent < 0)
            {
                for (s = exponent; s < 0; s++) result /=
            }
            return result;
        }
    }
}
exponent = (short)Math.Abs(exponent);
}
value = Round(value);
else
{
    while (FindSignificantsBeforeDecimal(separator, value) < significantsAfter)
    {
        constant *= 10;
        value *= 10;
    }
    value = Round(value)/constant;
}
// Power method cannot be used, as the exponentiated number may exceed the maximal long value.
exponent -= (short)(Math.Sign(dsExponent)*
    (FindSignificantDigits(separator, value, significantsAfter) - 1));
if (dsExponent != 0)
{
    String strValue = Convert.ToString(value, invC);
    short separatorAt = FindDecimalSeparatorPosition(strValue);
    if (separatorAt > -1)
    {
        strValue = strValue.Substring(0, separatorAt);
    }
    strValue += Common.EXPONENT + exponentSign
        + Convert.ToString(exponent);
    value = double.Parse(strValue, invC);
}
return value;
}

/// <summary>
/// Rounds a number according to mathematical rules.
/// </summary>
/// <param name="d">Number to be rounded</param>
/// <returns>Rounded number</returns>
public static double Round(double d)
{
    return (long)(d + .5);
}

/// <summary>
/// Converts a double value to a string such that it reflects the double format (without converting it to a scientific format by itself, as it is the case with Convert.ToString(double, invC)).
/// </summary>
/// <param name="d">Value to be converted</param>
/// <returns>Same format value as a string</returns>
public static String ConvertToString(double d)
double dValue = d;
StringBuilder value = new StringBuilder();

if (Math.Sign(dValue) == -1) value.Append(Common.DASH);
if ((dValue > 1E-5) && (dValue < 1E-4))
{
    value.Append("0");
    while ((long)dValue == 0)
    {
        dValue *= 10;
        if (dValue >= 1) break;
        value.Append(Convert.ToString((long)dValue));
    }
}
short exponent = FindExponent(d);
if (exponent != 0)
{
    value.Append(RetrieveDigits(dValue));
    InsertSeparator(dValue, value);
    value.Append(Common.EXPONENT);
    value.Append(exponent);
}
else
{
    value.Append(RetrieveDigits(dValue));
    InsertSeparator(dValue, value);
    if (value.Length > MAX_CHARACTERS + 3)
    {
        value.Length = MAX_CHARACTERS + 3;
    }
}
return value.ToString();

/// <summary>
/// Rounds to a fixed number of significant digits.
/// </summary>
/// <param name="d">Number to be rounded</param>
/// <param name="significantDigits">Requested number of significant digits</param>
/// <param name="separator">Language-specific decimal separator</param>
/// <returns>Rounded number</returns>
public static String RoundToString(char separator, double d, short significantDigits)
{
    // Number of significants that *are* before the decimal separator:
    short significantsBefore = FindSignificantsBeforeDecimal(separator, d);
    // Number of decimals that *should* be after the decimal separator:
    short significantsAfter = FindSignificantsAfterDecimal(significantsBefore, significantDigits);
    // Round to the specified number of digits after decimal
separator:
    double rounded = Maths.Round(separator, d,
    significantsAfter);

    String exponent = FindExponent(Convert.ToString(rounded,
    invC));
    String mantissa = FindMantissa(separator,
    Convert.ToString(rounded, invC));

double dMantissa = double.Parse(mantissa, invC);
StringBuilder result = new StringBuilder(mantissa);
// Determine the significant digits in this number:
short significants = FindSignificantDigits(separator,
    dMantissa,
    significantsAfter);
// Add lagging zeros, if necessary:
if (significants <= significantDigits)
{
    if (significantsAfter != 0)
    {
        result.Append(strZeros.Substring(0,
        CalculateMissingSignificantZeros(separator,
        dMantissa, significantsAfter)));
    }
    else
    {
        // Cut off the decimal separator & after decimal
digits:
        short decimalValue = (short)
        result.ToString().IndexOf(
        Convert.ToString(separator));
        if (decimalValue > -1) result.Length =
decimalValue;
    }
} else if (significantsBefore > significantDigits)
{
    d /= Power(10, (short)(significantsBefore -
significantDigits));
    d = Round(d);
    short digits = (short)(significantDigits + ((d < 0) ?
    1 : 0));
    String strD = d.ToString().Substring(0, digits);
    result.Length = 0;
    result.Append(strD + strZeros.Substring(0,
    significantsBefore - significantDigits));
}
if (short.Parse(exponent, invC) != 0)
{
    result.Append(Common.EXPONENT + exponent);
}
return result.ToString();
} // public static String RoundToString(...)
Extensive testing of a software is crucial for qualitative code. To say that the code is tested
does not give much information. The question is what is tested. Not in this case, but often
it is also important to know where (in which environment) it was tested, and how - i.e. the
test succession. Here is the code used to test the Maths class.

```csharp
using System;
using System.Collections.Generic;
namespace ConsoleApplicationCommons
{
    class TestCommon
    {
        /// <summary>
        /// Test for the common functionality
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            // Test rounding
            List<double> values = new List<double>();
            values.Add(0.0);
            AddValue(1.4012984643248202e-45, values);
            AddValue(1.999999757e-5, values);
            AddValue(1.999999757e-4, values);
            AddValue(1.999999757e-3, values);
            AddValue(0.000640589, values);
            AddValue(0.3396899998188019, values);
            AddValue(0.34, values);
            AddValue(7.07, values);
            AddValue(118.188, values);
            AddValue(118.2, values);
            AddValue(123.405009, values);
            AddValue(30.76994323730469, values);
            AddValue(130.76994323730469, values);
            AddValue(540, values);
            AddValue(12345, values);
            AddValue(123456, values);
            AddValue(540911, values);
            AddValue(9.223372036854776e56, values);
            const short SIGNIFICANTS = 5;
            foreach (double element in values)
            {
                + ",\" + Convert.ToString(element, Maths.invC) + ",
                + SIGNIFICANTS + ") = \" + Maths.RoundToString
                (Common.PERIOD, element, SIGNIFICANTS));
            }
        }
    }
}
```
```csharp
private static void AddValue(double d, List<double> values)
{
    values.Add(-d);
    values.Add(d);
}
```
If you are interested in a comparison with C++\(^5\), please compare it with the same example\(^6\) there. If you want to compare C# with Java\(^7\), take a look at the rounding number example\(^8\) there.

### 5.4 System.Windows.Forms

To create a Windows desktop application we use the library represented by `System.Windows.Forms` namespace. Some commonly used classes in this namespace include:

- Control\(^9\) - generic class from which other useful classes, like Form, TextBox and others listed below are derived
- Form\(^10\) - this is the base class for the program window. All other controls are placed directly onto a Form or indirectly on another container (like TabPage or TabControl) that ultimately resides on the Form. When automatically created in Visual Studio, it is usually subclassed as Form1.
- Button\(^11\) - a clickable button
- TextBox\(^12\) - a singleline or multiline textbox that can be used for displaying or inputting text
- RichTextBox\(^13\) - an extended TextBox that can display styled text, e.g. with parts of the text colored or with a specified font. RichTextBox can also display generalized RTF document, including embedded images.
- Label\(^14\) - simple control allowing display of a single line of unstyled text, often used for various captions and titles
- ListBox\(^15\) - control displaying multiple items (lines of text) with ability to select an item and to scroll through it
- ComboBox\(^16\) - similar to ListBox, but resembling a dropdown menu
- TabControl\(^17\) and TabPage\(^18\) - used to group controls in a tabbed interface (much like tabbed interface in Visual Studio or Mozilla Firefox). A TabControl contains a collection of TabPage objects.
5.5 Form class

The Form class (System.Windows.Forms.Form) is a particularly important part of that namespace because the form is the key graphical building block of 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:

```csharp
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.

5.6 Events

An event is an action being taken by the program when a user or the computer makes an action (for example, a button is clicked, a mouse rolls over an image, etc.). An event handler is an object that determines what action should be taken when an event is triggered.

```csharp
using System.Windows.Forms;
using System.Drawing;

public class ExampleForm : Form  // inherits from System.Windows.Forms.Form
{
    public ExampleForm()
    {
        this.Text = "I Love Wikibooks";  // specify title of the form
        this.Width = 300;  // width of the window in pixels
        this.Height = 300;  // height in pixels

        Button HelloButton = new Button();
        HelloButton.Location = new Point(20, 20);  // the location of button in pixels
    }
}
```

The .NET Framework

HelloButton.Size = new Size(100, 30); // the size of button in pixels
HelloButton.Text = "Click me!"; // the text of button

// When click in the button, this event fire
HelloButton.Click += new System.EventHandler(WhenHelloButtonClick);

this.Controls.Add(HelloButton);
}

void WhenHelloButtonClick(object sender, System.EventArgs e)
{
    MessageBox.Show("You clicked! Press OK to exit of this message");
}

public static void Main()
{
    Application.Run(new ExampleForm()); // display the form
}

5.7 Controls

The Windows Forms namespace has a lot of very interesting classes. One of the simplest and important is the Form class. A form is the key building block of any Windows application. It provides the visual frame that holds buttons, menus, icons and title bars together. Forms can be modal and modalless, owners and owned, parents and children. While forms could be created with a notepad, using a form editor like VS.NET, C# Builder or Sharp Develop makes development much faster. In this lesson, we will not be using an IDE. Instead, save the code below into a text file and compile with command line compiler.

using System.Windows.Forms;
using System.Drawing;

public class ExampleForm : Form // inherits from System.Windows.Forms.Form
{
    public ExampleForm()
    {
        this.Text = "I Love Wikibooks"; // specify title of the form
        this.BackColor = Color.White; // width of the window in pixels
        this.Width = 300;
        this.Height = 300; // height in pixels

        // A Label
        Label TextLabel = new Label();
        TextLabel.Text = "One Label here!";
        TextLabel.Location = new Point(20, 20);
        TextLabel.Size = new Size(150, 30);
        TextLabel.Font = new Font("Arial", 12); // See! we can modify the font of text
        this.Controls.Add(TextLabel); // adding the control to the form

        // A input text field
        TextBox Box = new TextBox(); // inherits from
        // display the form
    }
}
5.8 Lists

A list is a dynamic array that resizes itself as needed, if more data is inserted than it can hold at the time of insertion. Items can be inserted at any index, deleted at any index and accessed at any index. The C# non-generic list class is the ArrayList, while the generic one is List<T>.

Many of the List class' methods and properties are demonstrated in the following example:

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

namespace csharp_generic_list
{
    class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("List<T> demo");
            // Creating an instance that accepts strings
            List<string> foods = new List<string>();
            // Adding some items one by one with Add()
            foods.Add("bread");
            foods.Add("butter");
            foods.Add("chocolate");
            // Adding a simple string array with AddRange()
            string[] subList1 = {"orange", "apple", "strawberry", "grapes", "kiwi", "banana");
            foods.AddRange(subList1);
            // Adding another List<string> with AddRange()
            List<string> anotherFoodList = new List<string>();
            anotherFoodList.Add("yoghurt");
            anotherFoodList.Add("tomato");
            anotherFoodList.Add("roast beef");
            anotherFoodList.Add("vanilla cake");
            foods.AddRange(anotherFoodList);
            // Removing "orange" with Remove()
            foods.Remove("orange");
            // Removing the 5th (index = 4) item ("strawberry") with
            RemoveAt()
            foods.RemoveAt(4);
        }
    }
}
```
// Removing a range (4-7: all fruits) with
RemoveRange(int index, int count)
foods.RemoveRange(3, 4);

// sorting the list
foods.Sort();

// Printing the sorted foods
foreach (string item in foods)
{
    Console.Write("| " + item + " ");
}
Console.WriteLine("|");

// Removing all items from foods
foods.Clear();

// Printing the current item count in foods
Console.WriteLine("The list now has {0} items.",
foods.Count);
}
}

The terminal output is:

List<T> demo
| bread | butter | chocolate | roast beef | tomato | vanilla cake |
yoghurt |
The list now has 0 items.

5.9 LinkedLists

Items in a linked list can be accessed directly only one after the other. Of course an item at
any index can be accessed, but the list must iterate to the item from the first one, which is
much slower than accessing items by index in an array or a list. There is no non-generic
linked list in C#, while the generic one is LinkedList<T>.

5.10 Queues

A queue is a FIFO (first in - first out) collection. The item first pushed in the queue gets
taken first with the pop function. Only the first item is accessible at any time, and items
can only be put to the end. The non-generic queue class is called Queue, while the generic
one is Queue<T>.

5.11 Stacks

A stack is a LIFO (last in - first out) collection. The item pushed in first will be the last to
be taken by pop. Only the last item is accessible at any time, and items can only be put at
the top. The non-generic stack class is Stack, while the generic one is Stack<T>.
5.12 Hashtables and dictionaries

A dictionary is a collection of values with keys. The values can be very complex, yet searching the keys is still fast. The non-generic class is `Hashtable`, while the generic one is `Dictionary<TKey, TValue>`.

Threads are tasks that can run concurrently to other threads and can share data. When your program starts, it creates a thread for the entry point of your program, usually a `Main` function. So, you can think of a "program" as being made up of threads. The .NET Framework allows you to use threading in your programs to run code in parallel to each other. This is often done for two reasons:

1. If the thread running your graphical user interface performs time-consuming work, your program may appear to be unresponsive. Using threading, you can create a new thread to perform tasks and report its progress to the GUI thread.
2. On computers with more than one CPU or CPUs with more than one core, threads can maximize the use of computational resources, speeding up tasks.

5.13 The Thread class

The `System.Threading.Thread` class exposes basic functionality for using threads. To create a thread, you simply create an instance of the `Thread` class with a `ThreadStart` or `ParameterizedThreadStart` delegate pointing to the code the thread should start running. For example:

```csharp
using System;
using System.Threading;

class Program
{
    private static void SecondThreadFunction()
    {
        while (true)
        {
            Console.WriteLine("Second thread says hello.");
            Thread.Sleep(1000);  // pause execution of the current
            thread for 1 second (1000 ms)
        }
    }

class Main()
{
    Thread newThread = new Thread(new
    ThreadStart(SecondThreadFunction));
    newThread.Start();
    while (true)
    {
        Console.WriteLine("First thread says hello.");
        Thread.Sleep(500);  // pause execution of the current
    }
}
```

20 http://en.wikibooks.org/wiki/Category%3A
The .NET Framework

thread for half a second (500 ms)
}
}

You should see the following output:

First thread says hello.
Second thread says hello.
First thread says hello.
First thread says hello.
Second thread says hello.
...

Notice that the while keyword is needed because as soon as the function returns, the thread exits, or terminates.

5.13.1 ParameterizedThreadStart

The void ParameterizedThreadStart(object obj) delegate allows you to pass a parameter to the new thread:

```csharp
using System;
using System.Threading;

public static class Program
{
    private static void SecondThreadFunction(object param)
    {
        while (true)
        {
            Console.WriteLine("Second thread says " +
            param.ToString() + ".");
            Thread.Sleep(500); // pause execution of the current
            thread for half a second (500 ms)
        }
    }

    public static void Main()
    {
        Thread newThread = new Thread(new
        ParameterizedThreadStart(SecondThreadFunction));

        newThread.Start(1234); // here you pass a parameter to the
        new thread

        while (true)
        {
            Console.WriteLine("First thread says hello.");
            Thread.Sleep(1000); // pause execution of the current
            thread for a second (1000 ms)
        }
    }
}
```

The output is:
5.14 Sharing Data

Although we could use `ParameterizedThreadStart` to pass parameter(s) to threads, it is not typesafe and is clumsy to use. We could exploit anonymous delegates to share data between threads, however:

```csharp
using System;
using System.Threading;

public static class Program
{
    public static void Main()
    {
        int number = 1;
        Thread newThread = new Thread(new ThreadStart(delegate
        {
            while (true)
            {
                number++;
                Console.WriteLine("Second thread says " +
                    number.ToString() + ".");
                Thread.Sleep(1000);
            }
        }));
        newThread.Start();
        while (true)
        {
            number++;
            Console.WriteLine("First thread says " +
                number.ToString() + ".");
            Thread.Sleep(1000);
        }
    }
}
```

Notice how the body of the anonymous delegate can access the local variable `number`.

5.15 Asynchronous Delegates

Using anonymous delegates can lead to a lot of syntax, confusion of scope, and lack of encapsulation. However with the use of lambda expressions, some of these problems can be mitigated. Instead of anonymous delegates, you can use asynchronous delegates to pass and return data, all of which is type safe. It should be noted that when you use an asynchronous delegate, you are actually queueing a new thread to the thread pool. Also, using asynchronous delegates forces you to use the asynchronous model.
using System;

public static class Program
{
    delegate int del(int[] data);

    public static int SumOfNumbers(int[] data)
    {
        int sum = 0;
        foreach (int number in data) {
            sum += number;
        }
        return sum;
    }

    public static void Main()
    {
        int[] numbers = new int[] { 1, 2, 3, 4, 5 };
        del func = SumOfNumbers;
        IAsyncResult result = func.BeginInvoke(numbers, null, null);
        // I can do stuff here while numbers is being added
        int sum = func.EndInvoke(result);
        // sum = 15
    }
}

5.16 Synchronization

In the sharing data example, you may have noticed that often, if not all of the time, you will get the following output:

First thread says 2.
Second thread says 3.
Second thread says 5.
First thread says 4.
Second thread says 7.
First thread says 7.

One would expect that at least, the numbers would be printed in ascending order! This problem arises because of the fact that the two pieces of code are running at the same time. For example, it printed 3, 5, then 4. Let us examine what may have occurred:

1. After "First thread says 2", the first thread incremented number, making it 3, and printed it.
2. The second thread then incremented number, making it 4.
3. Just before the second thread got a chance to print number, the first thread incremented number, making it 5, and printed it.
4. The second thread then printed what number was before the first thread incremented it, that is, 4. Note that this may have occurred due to console output buffering.
The solution to this problem is to synchronize the two threads, making sure their code doesn't interleave like it did. C# supports this through the `lock` keyword. We can put blocks of code under this keyword:

```csharp
using System;
using System.Threading;

public static class Program
{
    public static void Main()
    {
        int number = 1;
        object numberLock = new object();
        Thread newThread = new Thread(new ThreadStart(delegate
        {
            while (true)
            {
                lock (numberLock)
                {
                    number ++;
                    Console.WriteLine("Second thread says " +
                    number.ToString() + ".");
                }
            }
        }));
        newThread.Start();
        while (true)
        {
            lock (numberLock)
            {
                number ++;
                Console.WriteLine("First thread says " +
                number.ToString() + ".");
            }
            Thread.Sleep(1000);
        }
    }
}
```

The variable `numberLock` is needed because the `lock` keyword only operates on reference types, not value types. This time, you will get the correct output:

```
First thread says 2.
Second thread says 3.
Second thread says 4.
First thread says 5.
Second thread says 6.
...
```

The `lock` keyword operates by trying to gain an exclusive lock on the object passed to it (`numberLock`). It will only release the lock when the code block has finished execution (that is, after the `}`). If an object is already locked when another thread tries to gain a lock on the same object, the thread will block (suspend execution) until the lock is released, and then lock the object. This way, sections of code can be prevented from interleaving.
5.16.1 Thread.Join()

The Join method of the Thread class allows a thread to wait for another thread, optionally specifying a timeout:

```csharp
using System;
using System.Threading;

public static class Program
{
    public static void Main()
    {
        Thread newThread = new Thread(new ThreadStart(delegate
        {
            Console.WriteLine("Second thread reporting.");
            Thread.Sleep(5000);
            Console.WriteLine("Second thread done sleeping.");
        }));
        newThread.Start();
        Console.WriteLine("Just started second thread.");
        newThread.Join(1000);
        Console.WriteLine("First thread waited for 1 second.");
        newThread.Join();
        Console.WriteLine("First thread finished waiting for second thread. Press any key.");
        Console.ReadKey();
    }
}
```

The output is:

```
Just started second thread.
Second thread reporting.
First thread waited for 1 second.
Second thread done sleeping.
First thread finished waiting for second thread. Press any key.
```

The .NET Framework currently supports calling unmanaged functions and using unmanaged data, a process called marshalling. This is often done to use Windows API functions and data structures, but can also be used with custom libraries.

5.17 GetSystemTimes

A simple example to start with is the Windows API function GetSystemTimes. It is declared as:

```csharp
BOOL WINAPI GetSystemTimes(
    __out_opt LPFILETIME lpIdleTime,
    __out_opt LPFILETIME lpKernelTime,
    __out_opt LPFILETIME lpUserTime
);
```

LPFILETIME is a pointer to a FILETIME structure, which is simply a 64-bit integer. Since C# supports 64-bit numbers through the long type, we can use that. We can then import and use the function as follows:
using System;
using System.Runtime.InteropServices;

public class Program
{
    [DllImport("kernel32.dll")]
    static extern bool GetSystemTimes(out long idleTime, out long kernelTime, out long userTime);

    public static void Main()
    {
        long idleTime, kernelTime, userTime;
        GetSystemTimes(out idleTime, out kernelTime, out userTime);
        Console.WriteLine("Your CPU(s) have been idle for: " + (new TimeSpan(idleTime)).ToString());
        Console.ReadKey();
    }
}

Note that the use of `out` or `ref` in parameters automatically makes it a pointer to the unmanaged function.

### 5.18 GetProcessIoCounters

To pass pointers to structs, we can use the `out` or `ref` keyword:

using System;
using System.Runtime.InteropServices;

public class Program
{
    struct IO_COUNTERS
    {
        public ulong ReadOperationCount;
        public ulong WriteOperationCount;
        public ulong OtherOperationCount;
        public ulong ReadTransferCount;
        public ulong WriteTransferCount;
        public ulong OtherTransferCount;
    }

    [DllImport("kernel32.dll")]
    static extern bool GetProcessIoCounters(IntPtr ProcessHandle, out IO_COUNTERS IoCounters);

    public static void Main()
    {
        IO_COUNTERS counters;
        Console.WriteLine("This process has read " + counters.ReadTransferCount.ToString("N0") + " bytes of data.");
        Console.ReadKey();
    }
}
6 Keywords

Abstract classes may contain abstract members in addition to implemented ones. That is, while some of the methods and properties in an abstract class may be implemented, others (the abstract members) may have their signatures defined, but have no implementation. Concrete subclasses derived from an abstract class define those methods and properties.

1 The `as` keyword casts an object to a different type. It is therefore similar to the `TypeA varA = (TypeA) varB` syntax. The difference is that this keyword returns null if the object was of an incompatible type, while the former method throws a type-cast exception in that case.

6.0.1 See also

- `is`

3 The keyword `base` describes that you would like to refer to the base class for the requested information, not in the current instantiated class.

A `base` class is the class in which the currently implemented class inherits from. When creating a class with no defined base class, the compiler automatically uses the `System.Object` base class.

Therefore the two declarations below are equivalent.

```csharp
public class MyClass
{
}

public class MyClass : System.Object
{
}
```

Some of the reasons the base keyword is used is:

- Passing information to the base class's constructor

```csharp
public class MyCustomException : System.Exception
{
    public MyCustomException() : base() {}
}
```

2 Chapter 6.1.4 on page 141
public MyCustomerException(string message, Exception innerException) : base(message, innerException) {}

// .......

• Recalling variables in the base class, where the newly implemented class is overriding its behaviour

public class MyBaseClass
{
    protected string className = "MyBaseClass";
}

public class MyNewClass : MyBaseClass
{
    protected new string className = "MyNewClass";

    public override string BaseClassName
    {
        get { return base.className; }
    }
}

• Recalling methods in the base class. This is useful when you want to add to a method, but still keep the underlying implementation.

// Necessary using's here

public class _Default : System.Web.UI.Page
{
    protected void InitializeCulture()
    {
        CultureInfo.GetSpecificCulture(Page.UICulture);

        base.InitializeCulture();
    }
}

4

The **bool** keyword is used in field, method, property, and variable declarations and in cast and typeof operations as an alias for the .NET Framework structure **System.Boolean**. That is, it represents a value of **true** or **false**. Unlike in C++, whose **boolean** is actually an **integer**, a **bool** in C# is its own data type and cannot be cast to any other primitive type.

7

The keyword **break** is used to exit out of a loop or switch block.

**break as used in a loop**

4 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
5 Chapter 3.2 on page 52
6 Chapter 3.5 on page 54
7 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
int x;
while (x < 20){
    if (x > 10) break;
    x++;
}

The while loop would increment x as long as it was less than twenty. However when x is incremented to ten the condition in the if statement becomes true, so the break statement causes the while loop to be broken and execution would continue after the closing parentheses.

**break as used in a switch block**

int x;

switch (x) {
    case 0:
        Console.WriteLine("x is 0");
        break;
    case 1:
        Console.WriteLine("x is 1");
        break;
    case 2:
        // falls through
    case 3:
        Console.WriteLine("x is 2 or 3");
        break;
}

When the program enters the switch block, it will search for a case statement that is true. Once it finds one, it will read any further statements printed until it finds a break statement. In the above example, if x is 0 or 1, the console will only print their respective values and then jump out of the statement. However, if the value of x is 2 or 3, the program will read the same proceeding statement(s) until it reaches a break statement. In order not to show anybody who reads the code that this handling for 2 is the same for three, it is good programming practice to add a comment like "falls through" after the falling-through cases.

The **byte** keyword is used in field, method, property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure **System.Byte**. That is, it represents an 8-bit unsigned integer whose value ranges from 0 to 255.

The keyword **case** is often used in a **switch** statement.

---

9  Chapter 3.2 on page 52
10 Chapter 3.5 on page 54
12 Chapter 6.1.4 on page 150
The keyword **catch** is used to identify a *statement or statement block* for execution, if an exception occurs in the body of the enclosing **try** block. The catch clause is preceded by the **try** clause, and may optionally be followed by a **finally** clause.

The **char** keyword is used in field, method, property, and variable declarations and in `cast` and `typeof` operations as an alias for the .NET Framework structure `System.Char`. That is, it represents a Unicode character whose from 0 to 65,535.

The **checked** and **unchecked** operators are used to control the overflow checking context for integral-type arithmetic operations and conversions. It checks, if there is an overflow (this is default).

**See also**


The **class** keyword is used to declare a class.

The **const** keyword is used in field and local variable declarations to make the variable constant. It is thus associated with its declaring class or assembly instead of with an instance of the class or with a method call. It is syntactically invalid to assign a value to such a variable anywhere other than its declaration.

**Further reading**

- [Constant function parameters](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%23Constant%20function%20parameters)

The keyword **continue** can be used inside any loop in a method. Its affect is to end the current loop iteration and proceed to the next one. If executed inside a for, end-of-loop statement is executed (just like normal loop termination).

---

13 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
14 Chapter 6.1.4 on page 151
15 Chapter 6.1.4 on page 151
16 Chapter 6.1.2 on page 136
17 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
18 Chapter 3.2 on page 52
19 Chapter 3.5 on page 54
20 http://en.wikibooks.org/wiki/%3Aw%3AUnicode
21 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
22 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
23 Chapter 3.1 on page 50
24 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
25 http://en.wikibooks.org/wiki/%3Aw%3AConstant%20%28programming%29%23Constant%20function%20parameters
26 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
The `decimal` keyword is used in field, method, property, and variable declarations and in `cast` and `typeof` operations as an alias for the .NET Framework structure `System.Decimal`. That is, it represents a signed, 128-bit decimal number whose value is 0 or a decimal number with 28 or 29 digits of precision ranging either from $-1.0 \times 10^{-28}$ to $-7.9 \times 10^{28}$ or from $1.0 \times 10^{-28}$ to $7.9 \times 10^{28}$.

The `default` keyword can be used in the `switch` statement or in generic code:

- The `switch` statement: Specifies the default label.
- Generic code: Specifies the default value of the type parameter. This will be `null` for reference types and zero for value types.

### 6.1 References

The `delegate` keyword is used to declare a `delegate`. A delegate is a programming construct that is used to obtain a callable reference to a method of a class.

The `do` keyword identifies the beginning of a `do ... ` loop.

The `double` keyword is used in field, method, property, and variable declarations and in `cast` and `typeof` operations as an alias for the .NET Framework structure `System.Double`. That is, it represents an IEEE 754, 64-bit signed binary floating point number whose value is `negative 0`, `positive 0`, `negative infinity`, `positive infinity`, `not a number`, or a number ranging either from $-5.0 \times 10^{-324}$ to $-1.79 \times 10^{308}$ or from $5.0 \times 10^{-324}$ to $1.79 \times 10^{308}$.

The `else` keyword identifies a `else` clause of an `if` statement with the following syntax:

---

28 Chapter 3.2 on page 52
29 Chapter 3.5 on page 54
30 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
31 `default` (C# Reference) 32. MSDN. Retrieved 2011-08-09 http://
34 http://msdn2.microsoft.com/en-us/library/xzth0h0d.aspx
35 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
36 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
37 Chapter 2.25.2 on page 35
38 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
39 Chapter 3.2 on page 52
40 Chapter 3.5 on page 54
41 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
42 Chapter 2.25.2 on page 35

133
if-statement ::= "if" "(" condition ")" if-body 'else' else-body
condition ::= boolean-expression
if-body ::= statement-or-statement-block
else-body ::= statement-or-statement-block

An else clause immediately follows an if-body. It provides 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 above example only checks whether myNumber is less than 0, if myNumber is not 4. It in turn only checks whether myNumber%2 is 0, if myNumber is not less than 0. Since none of the conditions are true, it executes the body of the final else clause.

The enum keyword identifies an enumeration44.

The event keyword is used to declare an event46.

43 http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords
44 Chapter 2.22 on page 31
45 http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords
46 Chapter 3.7 on page 55
47 http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords
6.1.1 General

When values are cast implicitly, the runtime does not need any casting in code by the developer in order for the value to be converted to its new type.

Here is an example, where the developer is casting explicitly:

```csharp
// Example of explicit casting.
float fNumber = 100.00f;
int iNumber = (int) fNumber;
```

The developer has told the runtime, 'I know what I'm doing, force this conversion.'

Implicit casting means that runtime doesn't need any prompting in order to do the conversion. Here is an example of this.

```csharp
// Example of implicit casting.
byte bNumber = 10;
in t iNumber = bNumber;
```

6.1.2 Keyword

Notice that no casting was necessary by the developer. What is special about implicit, is that the context that the type is converted to is totally lossless i.e. converting to this type loses no information, so it can be converted back without worry.

The `explicit` keyword is used to create type conversion operators that can only be used by specifying an explicit type cast.

This construct is useful to help software developers write more readable code. Having an explicit cast name makes it clear that a conversion is taking place.

```csharp
class Something
{
    public static explicit operator Something(string s)
    {
        // Convert the string to Something
    }
}
```

```csharp
string x = "hello";
// Implicit conversion (string to Something) generates a compile time error
Something s = x;

// This statement is correct (explicit type name conversion)
Something s = (Something) x;
```

The keyword `extern` indicates that the method being called exists in a DLL\(^49\).

---

\(^{48}\) [http://en.wikibooks.org/wiki/Category\%3AC\%20Sharp\%20Programming\%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)

\(^{49}\) [http://en.wikibooks.org/wiki/%3AW%3ADynamic-link%20library](http://en.wikibooks.org/wiki/%3AW%3ADynamic-link%20library)
A tool called \texttt{tlbimp.exe} can create a wrapper assembly that allows C# to interact with the DLL like it was a .NET assembly i.e. use constructors to instantiate it, call its methods.

Older DLLs will not work with this method. Instead, you have to explicitly tell the compiler what DLL to call, what method to call and what parameters to pass. Since parameter type is very important, you can also explicitly define what type the parameter should be passed to the method as.

Here is an example:

```csharp
using System;
using System.Runtime.InteropServices;

namespace ExternKeyword
{
    public class Program
    {
        static void Main()
        {
            NativeMethods.MessageBoxEx(IntPtr.Zero, "Hello there", "Caption here", 0, 0);
        }
    }

    public class NativeMethods
    {
        [DllImport("user32.dll")]
        public static extern MessageBoxEx(IntPtr hWnd, string lpText, string lpCaption, uint uType, short wLanguageId);
    }
}
```

The \texttt{[DllImport("user32.dll")]} tells the compiler which DLL to reference. Windows searches for files as defined by the PATH environment variable, and therefore will search those paths before failing.

The method is also static because the DLL may not understand how to be 'created', as DLLs can be created in different languages. This allows the method to be called directly, instead of being instantiated and then used.

The \texttt{false} keyword is a boolean constant value.

The keyword \texttt{finally} is used to identify a \textit{statement} or \textit{statement block} after a \texttt{try} - \texttt{catch} block for execution regardless of whether the associated try block encountered an exception, and executes even after a \texttt{return} statement. The finally block is used to perform cleanup activities.

\begin{itemize}
\item[50] \url{http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming}
\item[51] Chapter 6.0.1 on page 130
\item[52] \url{http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords}
\item[53] Chapter 6.1.4 on page 151
\item[54] Chapter 6.0.1 on page 132
\item[55] \url{http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords}
\end{itemize}
The **fixed** keyword is used to prevent the garbage collector from relocating a variable. You may only use this in an *unsafe* context.

```csharp
fixed (int *c = &shape.color) {
    *c = Color.White;
}
```

If you are using C# 2.0 or greater, the **fixed** may also be used to declare a fixed-size array. This is useful when creating code that works with a COM[^56] project or DLL[^57].

Your array must be composed of one of the primitive types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `ulong`, or `ushort`.

```csharp
protected fixed int monthDays[12];
```

The **float** keyword is used in field, method[^59], property[^60], and variable declarations and in `cast` and `typeof` operations as an alias for the .NET Framework structure `System.Single`. That is, it represents a IEEE 754, 32-bit signed binary floating point number whose value is `negative 0`, `positive 0`, `negative infinity`, `positive infinity`, `not a number`, or a number ranging either from \(-1.5 \times 10^{-45}\) to \(-3.4 \times 10^{38}\) or from \(1.5 \times 10^{-45}\) to \(3.4 \times 10^{38}\).

The **for** keyword identifies a **for** loop[^62].

The **foreach** keyword identifies a **foreach** loop[^64].

```csharp
// example of foreach to iterate over an array
public static void Main() {
    int[] scores = new int[] { 54, 78, 34, 88, 98, 12 };
    foreach (int score in scores) {
        total += score;
    }
    int averageScore = total/scores.length;
}
```

The **goto** keyword returns the flow of operation to the label which follows it. Labels can be created by putting a colon after any word. e.g.

[^56]: http://en.wikibooks.org/wiki/%3Aw%3AComponent%20Object%20Model
[^57]: http://en.wikibooks.org/wiki/%3Aw%3ADynamic-link%20Library
[^58]: http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
[^59]: Chapter 3.2 on page 52
[^60]: Chapter 3.5 on page 54
[^61]: http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
[^62]: Chapter 2.25.2 on page 35
[^63]: http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
[^64]: Chapter 2.25.2 on page 35
[^65]: http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
The use of goto is very controversial, because, when used frivolously, it creates code that jumps from place to place and is disorganized and hard to read. It is rarely even necessary because the same thing can often be accomplished with a more organized for loop or while loop.

The if keyword identifies an if statement with the following syntax:

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

If the condition evaluates to true, the if-body executes. Curly braces ('{' and '}') allow the if-body to contain more than one statement. 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:

```csharp
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 boolean expression used in an if statement typically contains one or more of the following operators:

66 Chapter 2.25.2 on page 35
67 Chapter 2.25.2 on page 35
68 [http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords)
69 Chapter 2.25.2 on page 35
6.1.3 General

When values are cast implicitly, the runtime does not need any casting in code by the developer in order for the value to be converted to its new type.

Here is an example, where the developer is casting explicitly:

```csharp
// Example of explicit casting.
float fNumber = 100.00f;
int iNumber = (int) fNumber;
```

The developer has told the runtime, "I know what I’m doing, force this conversion."

Implicit casting means that runtime doesn’t need any prompting in order to do the conversion. Here is an example of this.

```csharp
// Example of implicit casting.
byte bNumber = 10;
int iNumber = bNumber;
```

Notice that no casting was necessary by the developer. What is special about implicit is that the context that the type is converted to is totally lossless, i.e. converting to this type loses no information. So, it can be converted back without worry.

6.1.4 Keyword

The keyword `implicit` is used for a type to define how to can be converted implicitly. It is used to define what types can be converted to without the need for explicit casting.

As an example, let us take a Fraction class, that will hold a nominator (the number at the top of the division), and a denominator (the number at the bottom of the division). We will add a property so that the value can be converted to a `float`.

```csharp
public class Fraction
{
    private int nominator;
    private int denominator;
}
```
public Fraction(int nominator1, int denominator1)
{
    nominator = nominator1;
    denominator = denominator1;
}

tenominator = nominator1;

denominator = denominator1;

public float Value { get { return
(float)_nominator/(float)_denominator; } }

public static implicit operator float(Fraction f)
{
    return f.Value;
}

public override string ToString()
{
    return _nominator + "/" + _denominator;
}

public class Program
{
    [STAThread]
    public static void Main(string[] args)
    {
        Fraction fractionClass = new Fraction(1, 2);
        float number = fractionClass;
        Console.WriteLine("(0) = (1)", fractionClass, number);
    }
}

To re-iterate, the value it implicitly casts to must hold data in the form that the original
class can be converted back to. If this is not possible, and the range is narrowed (like
converting double to int), use the explicit operator.

The in keyword identifies the collection to enumerate in a foreach loop.

The int keyword is used in field, method, property, and variable declarations and in
cast and typeof operations as an alias for the .NET Framework structure System.Int32.
That is, it represents a 32-bit signed integer whose value ranges from -2,147,483,648 to
2,147,483,647.

The interface keyword is used to declare an interface. Interfaces provide a construct for
a programmer to create types that can have methods, properties, delegates, events, and
indexers declared, but not implemented.

---

73 Chapter 2.25.2 on page 35
74 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
75 Chapter 3.2 on page 52
76 Chapter 3.5 on page 54
77 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
It is a good programming practice to give interfaces differing names from classes that start with an *I* and/or finish with *...able*, like *IRun* or *Runnable* or *IRunnable*.

The **internal** keyword is an *access modifier* used in field, method79, and property80 declarations to make the field, method, or property *internal* to its enclosing assembly. That is, it is only visible81 within the assembly that implements it.

The **is** keyword compares an object to a type, and if they're the same or of the same 'kind' (the object inherits83 the type), returns **true**. The keyword is therefore used to check for type compatibility, usually before *casting* (converting) a source type to a destination type in order to ensure that won't cause a type-cast exception to be thrown. Using **is** on a null variable always returns **false**.

This code snippet shows a sample usage:

```csharp
System.IO.StreamReader reader = new StreamReader("readme.txt");
bool b = reader is System.IO.TextReader;
// b is now set to true, because StreamReader inherits TextReader
```

The **lock** keyword allows a section of code to exclusively use a resource, a feature useful in multi-threaded applications. If a lock to the specified object is already held when a piece of code tries to lock the object, the code's thread is blocked until the object is available.

```csharp
using System;
using System.Threading;

class LockDemo
{
    private static int number = 0;
    private static object lockObject = new object();

    private static void DoSomething()
    {
        while (true)
        {
            lock (lockObject)
            {
                int originalNumber = number;
                number += 1;
                Thread.Sleep((new Random()).Next(1000)); // sleep for a random amount of time
                number += 1;
            }
        }
    }
}
```

---

78 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
79 Chapter 3.2 on page 52
80 Chapter 3.5 on page 54
82 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
83 Chapter 4 on page 73
84 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
Thread.Sleep((new Random()).Next(1000)); // sleep again

Console.Write("Expecting number to be " +
(originalNumber + 2).ToString());
Console.WriteLine(" and it is: " +
number.ToString());
// without the lock statement, the above would
produce unexpected results,
// since the other thread may have added 2 to the
number while we were sleeping.
}
}

public static void Main()
{
    Thread t = new Thread(new ThreadStart(DoSomething));
    t.Start();
    DoSomething(); // at this point, two instances of DoSomething
    are running at the same time.
}

The parameter to the lock statement must be an object reference, not a value type:

class LockDemo2
{
    private int number;
    private object obj = new object();

    public void DoSomething()
    {
        lock (this) // ok
        {
            ...
        }

        lock (number) // not ok, number is not a reference
        {
            ...
        }

        lock (obj) // ok, obj is a reference
        {
            ...
        }
    }
}

The long keyword is used in field, method\[^{86}\], property\[^{87}\], and variable declarations and in cast
and typeof operations as an alias for the .NET Framework structure System.Int64. That
is, it represents a 64-bit signed integer whose value ranges from -9,223,372,036,854,775,808
to 9,223,372,036,854,775,807.

[^85]: http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
[^86]: Chapter 3.2 on page 52
[^87]: Chapter 3.5 on page 54
[^88]: http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
The **namespace** keyword is used to supply a *namespace* for class, structure, and type declarations.

89

The **new** keyword has two different meanings:

1. It is an operator that requests a new instance of the class identified by its argument.
2. It is a modifier that explicitly hides a member.

As an example, see the code below:

```csharp
public class Car
{
    public void go()
    {
    }
}

Car theCar = new Car(); // The new operator creates a Car instance
int i = new int(); // Identical to ... = 0;

public class Lamborghini : Car
{
    public new void go() // Hides Car.go() with this method
    {
    }
}
```

The **null** keyword represents an empty value for a *reference* type variable, i.e. for a variable of any type derived from `System.Object`. In C# 2.0, **null** also represents the empty value for nullable *value* type variables.

90

The **object** keyword is used in field, method, property, and variable declarations and in *cast* and *typeof* operations as an alias for the .NET Framework structure `System.Object`. That is, it represents the base class from which all other *reference types* derive. On some platforms, the size of the reference is 32 bits, while on other platforms it is 64 bits.

91

The **operator** keyword allows a class to overload arithmetic and cast operators:

```csharp
public class Complex
{
    private double re, im;
    public double Real
```

References

89  [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
90  [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
91  [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
92  Chapter 3.2 on page 52
93  Chapter 3.5 on page 54
94  [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)

143
{  
    get { return re; }  
    set { re = value; }  
}

public double Imaginary  
{  
    get { return im; }  
    set { im = value; }  
}

// binary operator overloading  
public static Complex operator +(Complex c1, Complex c2)  
{  
    return new Complex() { Real = c1.Real + c2.Real, Imaginary = c1.Imaginary + c2.Imaginary };  
}

// unary operator overloading  
public static Complex operator -(Complex c)  
{  
    return new Complex() { Real = -c.Real, Imaginary = -c.Imaginary };  
}

// cast operator overloading (both implicit and explicit)  
public static implicit operator double(Complex c)  
{  
    // return the modulus: sqrt(x^2 + y^2)  
    return Math.Sqrt(Math.Pow(c.Real, 2) + Math.Pow(c.Imaginary, 2));  
}

public static explicit operator string(Complex c)  
{  
    // we should be overloading the ToString() method, but this  
    // is just a demonstration  
    return c.Real.ToString() + " + " + c.Imaginary.ToString() + "i";  
}

public class StaticDemo  
{  
    public static void Main()  
    {  
        Complex number1 = new Complex() { Real = 1, Imaginary = 2 };  
        Complex number2 = new Complex() { Real = 4, Imaginary = 10 };  
        Complex number3 = number1 + number2; // number3 now has Real  
        number3 = -number3; // number3 now has Real = -5, Imaginary =  
        double testNumber = number3; // testNumber will be set to the  
        // The cast to string was needed because that was an explicit  
        // cast operator.  
    }  
}

95  

95 http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords
The **out** keyword explicitly specifies that a variable should be passed *by reference* to a method, and set in that method. A variable using this keyword must *not* be initialized before the method call to ensure the developer understand its intended effects. Using this keyword requires the called method to set the variable using this modifier before returning. Using **out** also requires the developer to specify the keyword even in the calling code, to ensure that it is easily visible to developers reading the code that the variable will have its value changed elsewhere, which is useful when analyzing the program flow.

An example of passing a variable with **out** follows:

```csharp
void CallingMethod()
{
    int i;
    SetDependingOnTime(out i);
    // i is now 10 before/at 12 am, or 20 after
}

void SetDependingOnTime(out int iValue)
{
    iValue = DateTime.Now.Hour <= 12 ? 10 : 20;
}
```

The keyword **override** is use in declaring an overridden function, which extends a base class function of the same name.

**Further reading**

- Inheritance keywords\(^{97}\)

The keyword **params** is used to describe when a grouping of parameters are passed to a method, but the number of parameters are not important, as they may vary. Since the number isn't important, the **params** keyword must be the last variable in a method signature so that the compiler can deal with the parameters which have been defined first, before dealing with the params.

Here are examples of where it will, and will not work:

```csharp
// This works
public static void AddToShoppingBasket(decimal total, params string[] items)
{
    // ....
}

// This works
public static void AddToShoppingBasket(decimal total, int totalQuantity, params string[] items)
{
    // ....
}
```

\(^{96}\) [http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords)  
\(^{97}\) Chapter 4.5 on page 77  
\(^{98}\) [http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords)
// THIS DOES NOT WORK
public static void AddToShoppingBasket(params string[] items, decimal total, int totalQuantity)
{
    // ....
}

A good example of this is the \texttt{String.Format} method. The \texttt{String.Format} method allows a user to pass in a string formatted to their requirements, and then lots of parameters for the values to insert into the string. Here is an example:

\begin{verbatim}
public static string FormatMyString(string format, params string[] values)
{
    string myFormat = "Date: {0}, Time: {1}, WeekDay: {2}"
    return String.Format(myFormat, DateTime.Now.ToShortDateString(),
                        DateTime.Now.ToShortTimeString(), DateTime.Now.DayOfWeek);
}
\end{verbatim}

// Output will be something like:
// // Date: 7/8/2007, Time: 13:00, WeekDay: Tuesday;
// //

The \texttt{String.Format} method has taken a string, and replaced the \{0\}, \{1\}, \{2\} with the 1st, 2nd and 3rd parameters. If the \texttt{params} keyword did not exist, then the \texttt{String.Format()} would need an infinite number of overloads to cater for each case.

\begin{verbatim}
public string Format(string format, string param1)
{
    // ....
}
public string Format(string format, string param1, string param2)
{
    // ....
}
public string Format(string format, string param1, string param2, string param3)
{
    // ....
}
public string Format(string format, string param1, string param2, string param3, string param4)
{
    // ....
}
public string Format(string format, string param1, string param2, string param3, string param4, string param5)
{
    // ....
}
// To infinitum
\end{verbatim}

\footnote{http://en.wikibooks.org/wiki/Category\%3A\%20Sharp\%20Programming}
The `private` keyword is used in field, method\textsuperscript{100}, and property\textsuperscript{101} declarations to make the field, method, or property private to its enclosing class. That is, it is not visible\textsuperscript{102} outside of its class.

The `protected` keyword is used in field, method\textsuperscript{104}, and property\textsuperscript{105} declarations to make the field, method, or property protected to its enclosing class. That is, it is not visible\textsuperscript{106} outside of its class.

The `public` keyword is used in field, method\textsuperscript{108}, and property\textsuperscript{109} declarations to make the field, method, or property public to its enclosing class. That is, it is visible\textsuperscript{110} from any class.

The `readonly` keyword is closely related to the `const` keyword at a glance, with the exception of allowing a variable with this modifier to be initialized in a constructor, along with being associated with a class instance (object) rather than the class itself.

The primary use for this keyword is to allow the variable to take on different values depending on which constructor was called, in case the class has many, while still ensuring the developer that it can never intentionally or unintentionally be changed in the code once the object has been created.

This is a sample usage, assumed to be in a class called `SampleClass`:

```csharp
readonly string s;
SampleClass()
{
    s = "Hello!";
}
```

The `ref` keyword explicitly specifies that a variable should be passed by reference rather than by value.
A developer may wish to pass a variable by reference particularly in case of value types. If a variable is passed by reference, only a pointer is sent to a function in reality, reducing the cost of a method call in case it would involve copying large amounts of data, something C# does when normally passing value types.

Another common reason to pass a variable by reference is to let the called method modify its value. Because this is allowed, C# always enforces specifying that a value is passed by reference even in the method call, something many other programming languages don’t. This let developers reading the code easily spot places that can imply a type has had its value changed in a method, which is useful when analyzing the program flow.

Passing a value by reference does not imply that the called method has to modify the value; see the out keyword for this.

Passing by reference requires the passed variable to be initialized.

An example of passing a variable by reference follows:

```csharp
void CallingMethod()
{
    int i = 24;
    if (DoubleIfEven(ref i))
        Console.WriteLine("i was doubled to {0}", i); // outputs "i was doubled to 48"
}

bool DoubleIfEven(ref int iValue)
{
    if (iValue%2 == 0)
    {
        iValue *= 2;
        return true;
    }
    return false;
}
```

The return keyword is used to return execution from a method or from a property accessor. If the method or property accessor has a return type, the return keyword is followed by the value to return.

The sbyte keyword is used in field, method, property, and variable declarations and in cast and typeof operations as an alias for the .NET Framework structure System.SByte. That is, it represents an 8-bit signed integer whose value ranges from -128 to 127.

113 Chapter 2.9 on page 17
114 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
115 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
116 Chapter 3.2 on page 52
117 Chapter 3.5 on page 54
118 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
The `sealed` keyword is used to specify that a class cannot be inherited from. The following example shows the context in which it may be used:

```csharp
public sealed class
{
    ...
}
```

Notice: The `sealed` class inheritance is the same as that of a `final` class in Java.

The `short` keyword is used in field, method\(^{120}\), property\(^{121}\), and variable declarations and in `cast` and `typeof` operations as an alias for the .NET Framework structure `System.Int16`. That is, it represents a 16-bit signed integer whose value ranges from -32,768 to 32,767.

The `sizeof` keyword returns how many bytes an object requires to be stored.

An example usage:

```csharp
int i = 123456;
Console.WriteLine("Storing i, a (0), requires (1) bytes, or (2) bits.",
i.GetType(), sizeof(i), sizeof(i)*8);
// outputs "Storing i, a System.Int32, requires 4 bytes, or 32 bits."
```

The keyword `alloca` is used in an unsafe code context to allocate a block of memory on the stack.

```csharp
int* fib = stackalloc int[100];
```

In the example above, a block of memory of sufficient size to contain 100 elements of type `int` is allocated on the stack, not the heap; the address of the block is stored in the pointer `fib`. This memory is not subject to garbage collection and therefore does not have to be pinned (via `fixed`). The lifetime of the memory block is limited to the lifetime of the method in which it is defined (there is no way to free the memory before the method returns).

`alloca` is only valid in local variable initializers.

Because `Pointer` types are involved, `alloca` requires unsafe context. See Unsafe Code and Pointers.

`alloca` is similar to `_alloca` in the C run-time library.

---

\(^{119}\) [http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords)

\(^{120}\) Chapter 3.2 on page 52

\(^{121}\) Chapter 3.5 on page 54

\(^{122}\) [http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords)

\(^{123}\) [http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords)
The `static` keyword is used to declare a `class` or a class member (`method`, `property`, `field`, or `variable`) as `static`. A `class` that is declared `static` has only `static` members, and these are associated with the entire class instead of class `instances`.

The `string` keyword is used in field, method, property, and variable declarations and in `cast` and `typeof` operations as an alias for `System.String`. That is, it indicates an immutable sequence of characters.

The `struct` keyword declares a structure, i.e. a `value type` that functions as a light-weight `class`.

The `switch` statement is a control statement that handles multiple selections and enumerations by passing control to one of the case statements within its body.

This is an example of a switch statement:

```csharp
int currentAge = 18;
switch currentAge {
    case 16:
        Console.WriteLine("You can drive!\n")
        break;
    case 18:
        Console.WriteLine("You're finally an adult!\n");
        break;
    default:
        Console.WriteLine("Nothing exciting happened this year.\n");
        break;
}
```

Console Output
You're finally an adult!
The **this** keyword is used in an *instance method* or *instance property* to refer to the *current* class instance. That is, **this** refers to the object through which its containing method or property was invoked. It is also used to define extension methods.

The **throw** keyword is used to *throw* an exception object.

The **true** keyword is a Boolean constant value. Therefore

```csharp
while(true)
```

would create an infinite loop.

The **try** keyword is used to identify a *statement* or *statement block* as the body of an exception handling sequence. The body of the exception handling sequence must be followed by a **catch** clause, a **finally** clause, or both.

```csharp
try
{
    foo();
} catch(Exception Exc)
{
    throw new Exception("this is the error message", Exc);
}
```

The **typeof** keyword returns an instance of the **System.Type** class when passed a name of a class. It is similar to the **sizeof** keyword in that it returns a value instead of starting a section (block) of code (see **if**, **try**, **while**).

An example:

```csharp
using System;

namespace MyNamespace
{
    class MyClass
    {
        static void Main(string[] args)
        {
            Type t = typeof(int);
            Console.Out.WriteLine(t.ToString());
        }
    }
}
```

---

132 Chapter 4.17 on page 90
133 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
134 Chapter 6.0.1 on page 130
135 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
136 Chapter 6.0.1 on page 132
137 Chapter 6.1.2 on page 136
138 Chapter 6.1.4 on page 149
139 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
140 Chapter 6.1.4 on page 149
Keywords

```csharp
    Console.In.Read();
    
    System.Int32

    }  
    
    }  

The output will be:

```csharp
System.Int32

```

It should be noted that unlike `sizeof`, only class names themselves and not variables can be passed to `typeof` as shown here:

```csharp
using System;

namespace MyNamespace
{
    class MyClass2
    {
        static void Main(string[] args)
        {
            char ch;
            // This line will cause compilation to fail
            Type t = typeof(ch);
            Console.Out.WriteLine(t.ToString());
            Console.In.Read();
        }
    }
}
```

Sometimes, classes will include their own `GetTypeInfo()` method that will be similar, if not identical, to `typeof`.

141

The `uint` keyword is used in field, method\(^{142}\), property\(^{143}\), and variable declarations and in cast and `typeof` operations as an alias for the .NET Framework structure `System.UInt32`. That is, it represents a 32-bit unsigned integer whose value ranges from 0 to 4,294,967,295.

144

The `ulong` keyword is used in field, method\(^{145}\), property\(^{146}\), and variable declarations and in cast and `typeof` operations as an alias for the .NET Framework structure `System.UInt64`. That is, it represents a 64-bit unsigned integer whose value ranges from 0 to 18,446,744,073,709,551,615.

147
The `unchecked` keyword prevents overflow-checking when doing integer arithmetics. It may be used as an operator on a single expression or as a statement on a whole block of code.

```csharp
int x, y, z;
x = 1221111000;
y = 1221111000;

// used as an operator
z = unchecked(x*y);

// used as a statement
unchecked {
    z = x*y;
    x = z*z;
}
```

The `unsafe` keyword may be used to modify a procedure or define a block of code which uses unsafe code. Code is unsafe if it uses the "address of" (`&`) or pointer operator (`*`). In order for the compiler to compile code containing this keyword, you must use the `unsafe` option when using the Microsoft C-Sharp Compiler.

```csharp
// example of unsafe to modify a procedure
class MyClass {
    unsafe static void(string *msg) {
        Console.WriteLine(*msg);
    }
}

// example of unsafe to modify a code block
string s = "hello";
unsafe {
    char *cp = &s[2];
    *cp = 'a';
}
```

The `ushort` keyword is used in field, method, property, and variable declarations and in cast and `typeof` operations as an alias for the .NET Framework structure `System.UInt16`. That is, it represents a 16-bit unsigned integer whose value ranges from 0 to 65,535.

The `using` keyword has two completely unrelated meanings in C#, depending on if it is used as a directive or a statement.

---

148 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
149 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
150 Chapter 3.2 on page 52
151 Chapter 3.5 on page 54
152 http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords
6.2 The directive

**using** as a *directive* resolves unqualified type references so that a developer doesn't have to specify the complete namespace.

Example:

```csharp
using System;

// A developer can now type ,Console.WriteLine();, rather than
// ,System.Console.WriteLine();,

using can also provide a *namespace alias* for referencing types.

Example:

```csharp
using utils = Company.Application.Utilities;
```

6.3 The statement

**using** as a *statement* automatically calls the *dispose* on the specified object. The object must implement the *IDisposable* interface. It is possible to use several objects in one statement as long as they are of the same type.

Example:

```csharp
using (System.IO.StreamReader reader = new
    StreamReader("readme.txt"))
{
    // read from the file
}

// The file readme.txt has now been closed automatically.

using (Font headerFont = new Font("Arial", 12.0f),
    textFont = new Font("Times New Roman", 10.0f))
{
    // Use headerFont and textFont.
}

// Both font objects are closed now.
```

The **var** keyword can be used in place of a type when declaring a variable to allow the compiler to infer the type of the variable. This feature can be used to shorten variable declarations, especially when instantiating generic types, and is even necessary with LINQ\(^{154}\) expressions (since queries may generate very complex types).

The following:

```csharp
int num = 123;
```

---

153  http://en.wikibooks.org/wiki/Category%3A%20Sharp%20Programming%20Keywords
The statement

```csharp
string str = "asdf";
Dictionary<int, string> dict = new Dictionary<int, string>();
```

is equivalent to:

```csharp
var num = 123;
var str = "asdf";
var dict = new Dictionary<int, string>();
```

**var** does not create a "variant" type; the type is simply inferred by the compiler. In situations where the type cannot be inferred, the compiler generates an error:

```csharp
var str; // no assignment, can't infer type
void Function(var arg1, var arg2) // can't infer type
{
}
```

The keyword **virtual** is applied to a method declaration to indicate that the method may be overridden in a subclass. If the **virtual** keyword is not applied and a method is defined in a subclass with the same signature as the one in the parent class, the method in the parent class is hidden by the subclass implementation. With other words, it is only possible to have a true polymorphism**156** of functions with this keyword.

Notice: Comparing it with Java**157**, a method is not virtual if and only if it is **final**. This is the result of different design philosophies**158**.

The **void** keyword is used in method signatures to declare a method that does not return a value. A method declared with the **void** return type cannot provide any arguments to any **return** statements they contain.

Example:

```csharp
public void WorkRepeatedly(int numberOfTimes)
{
    for(int i = 0; i < numberOfTimes; i++)
        if(EarlyTerminationIsRequested)
            return;
        else
            DoWork();
}
```

**155** [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
**156** [http://en.wikibooks.org/wiki/%3Aw%3APolymorphism%20%28computer%20science%29](http://en.wikibooks.org/wiki/%3Aw%3APolymorphism%20%28computer%20science%29)
**158** [http://en.wikibooks.org/wiki/%3Aw%3AVirtual%20function%23The%20virtual%20philosophy](http://en.wikibooks.org/wiki/%3Aw%3AVirtual%20function%23The%20virtual%20philosophy)
**160** Chapter 3.2 on page 52
**161** [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
Keywords

The `volatile` keyword is used to declare a variable that may change its value over time due to modification by an outside process, the system hardware, or another concurrently running thread.

You should use this modifier in your member variable declaration to ensure that whenever the value is read, you are always getting the most recent (up-to-date) value of the variable.

```csharp
class MyClass
{
    public volatile long systemclock;
}
```

This keyword has been part of the C# programming language since .NET Framework 1.1 (Visual Studio 2003).

The `while` keyword identifies a `while` loop.

Special C# Identifiers

The `add` and `remove` keywords allow you to execute code whenever a delegate is added or removed from an event. Its usage is similar to the `get` and `set` keywords with properties:

```csharp
public event MyDelegateType MyEvent
{
    add
    {
        // here you can use the keyword "value" to access the delegate that is being added
        ... 
    }
    remove
    {
        // here you can use the keyword "value" to access the delegate that is being removed
        ... 
    }
}
```

The code in the `add` block will be executed when a delegate is added to the event. Similarly, the code in the `remove` block will be executed when a delegate is removed from the event.

The `alias` keyword is used to indicate an `external alias`.

When you need to use several versions of the same assembly or assemblies with the same full qualified typenames, you need to use the `alias` and `extern` keywords to give different alias names for each version.

Example:

162 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
163 Chapter 2.25.2 on page 35
164 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
165 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)

156
extern alias AppTools;
extern alias AppToolsV2;

To use the typenames of each version, you have the operator :: .

Example:

AppTools::MainTool tool_v1 = new AppTools::MainTool();
AppToolsV2::MainTool tool_v2 = new AppToolsV2::MainTool();

However, this only says to the compiler that there are several assemblies with typename conflicts. To relate what of each assemblies match's the alias name, you have to tell the compiler on its options apart the source. On dotNet command line, this options would be:

/r:AppTools=AppToolsv100.dll /r:AppToolsV2=AppToolsv200.dll

Notice: In order for it to be of use, you need to provide an external assembly to the compiler (e.g. pass /r:EXTALIAS=XXX.dll) and identify the external alias within the code (e.g. extern alias EXTALIAS;)

The special identifier get is used to declare the read accessor for a property.

The global keyword is useful in some contexts to resolve ambiguity between identifiers. If you have a conflict between a class name and a namespace, for example, you can use the global keyword to access the namespace:

namespace MyApp
{
    public static class System
    {
        public static void Main()
        {
            global::System.Console.WriteLine("Hello, World!");
            // if we had just used System.Console.WriteLine,
            // the compile would think that we referred to a
            // class named "Console" inside our "System" class.
        }
    }
}

global does not work in the following situation, however, as our System class does not have a namespace:

public static class System
{
    public static void Main()
    {
        global::System.Console.WriteLine("Hello, World!");
        // "System" doesn't have a namespace, so the above
        // would be referring to this class!
    }
}

166 http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords
167 http://en.wikibooks.org/wiki/Category%3ASharp%20Programming%20Keywords
The special identifier `partial` is used to allow `developers` to build `classes` from different files and have the compiler generate one class, combining all the partial classes. This is mostly useful for separating classes into separate blocks. For example, Visual Studio 2005 separates the UI code for forms into a separate partial class that allows you to work on the business logic separately.

The special identifier `set` is used to declare the `write accessor` for a `property`.

The special identifier `value` is used in a property's `write accessor` to represent the value requested for assignment to the `property`.

The `where` keyword has two different meanings:

1. It is used to specify one or more constraints on generic type parameters.
2. With LINQ, it is used to query a data source and select or filter elements to return.

The `yield` keyword returns the next value from an iterator or ends an iteration.

### 6.4 References

170 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
171 [http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords](http://en.wikibooks.org/wiki/Category%3AC%20Sharp%20Programming%20Keywords)
172 Chapter 4.16 on page 89
175 Chapter 2.25.2 on page 35
7 Contributors

<table>
<thead>
<tr>
<th>Edits</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Adrignola^1</td>
</tr>
<tr>
<td>1</td>
<td>Alexcey^2</td>
</tr>
<tr>
<td>2</td>
<td>Arbitrary^3</td>
</tr>
<tr>
<td>1</td>
<td>Astillman^7</td>
</tr>
<tr>
<td>9</td>
<td>Avicennasis^5</td>
</tr>
<tr>
<td>1</td>
<td>Bacon^6</td>
</tr>
<tr>
<td>7</td>
<td>Beno1000^7</td>
</tr>
<tr>
<td>1</td>
<td>Bombe^8</td>
</tr>
<tr>
<td>2</td>
<td>Buttink^9</td>
</tr>
<tr>
<td>2</td>
<td>C.Bharati^10</td>
</tr>
<tr>
<td>2</td>
<td>Cbmeeks^11</td>
</tr>
<tr>
<td>1</td>
<td>Chowmeined^12</td>
</tr>
<tr>
<td>1</td>
<td>Crazycomputers^13</td>
</tr>
<tr>
<td>2</td>
<td>Cristobaljbor^14</td>
</tr>
<tr>
<td>24</td>
<td>Darklama^15</td>
</tr>
<tr>
<td>1</td>
<td>David C Walls^16</td>
</tr>
<tr>
<td>5</td>
<td>Ddas^17</td>
</tr>
<tr>
<td>1</td>
<td>Derbeth^18</td>
</tr>
<tr>
<td>1</td>
<td>Devourer09^19</td>
</tr>
<tr>
<td>5</td>
<td>Dirk Hünniger^20</td>
</tr>
<tr>
<td>9</td>
<td>Dm7475^21</td>
</tr>
</tbody>
</table>

Contributors

38 Dominicz82
1 DuLithgow
1 Dunstvangeet
1 Dzikasosna
7 Eray
1 Fatcat1111
2 Feraudyh
15 FewEditsToday
1 Fishpi
5 Fly4fun
7 Forage
3 Frank (Usurped)
4 Gandalfxviv
1 Gbs256
3 Gorpik
1 Gotovikas
36 GreenVoid
1 HGatta
4 Ha98574
2 Hagindaz
15 Herbythyme
7 Hethrir
1 HethrirBot
3 Huan086
2 HumanThePequenino

References

Contributors

1 Mortense
3 Mukeshnt
2 Mwtoews
2 N1mxv
1 Nanodeath
8 Nercy
3 Netboy2005
2 Nfgdayton
16 Northgrove
1 Nvineeth
83 Ohms law
2 Onlyforu37
2 Orion Blastar
7 Panic2k4
2 Pcu123456789
3 Peachpuff
2 Phil.a
9 Phyll Chloro
2 Plee
2 Polluks
1 Purnil
1 QUBot
8 QuiteUnusual
2 Ramac
11 Recent Runes

7 Ripper234
214 Rodasmith
1 S.Örvarr.S
203 Sae1962
1 Scorchsaber
61 Sigma 7
1 Soeb
1 Spiderman
3 Spongebob88
1 Swatkatz14
1 Szelee
6 Tetsuo86
5 Thambiduraip
1 Tom Morris
1 Vaile
2 Vito Genovese
3 Watcher
2 Weblum
1 Whiteknight
2 Withinfocus
71 Wj32
1 Wutsje
1 Xania
3 Xraytux
2 Yurik

1 Zr40
2 שוחדופיק

http://en.wikibooks.org/w/index.php?title=User:%D7%A7%D7%99%D7%A4%D7%95%D7%93%D7%93%D7%A9
List of Figures

- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- CFR: Copyright free use.

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses\textsuperscript{124}. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrowser.

\textsuperscript{124} Chapter 8 on page 169
This License refers to version 3 of the GNU General Public License.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU FDL. (If you have received this document from someone other than the author, you may need to replace the FDL with a suitable local license.) This does not automatically mean that all of the content here is copyright free. Please check the content carefully to ensure that you are permitted to use it. In some cases, content is licensed under a Creative Commons license. Please check the content carefully to ensure that you are permitted to use it.

This License applies to each separately copyrighted work.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU FDL. (If you have received this document from someone other than the author, you may need to replace the FDL with a suitable local license.) This does not automatically mean that all of the content here is copyright free. Please check the content carefully to ensure that you are permitted to use it. In some cases, content is licensed under a Creative Commons license. Please check the content carefully to ensure that you are permitted to use it.
portion of this contract, and you cure the violation, to use that work under the conditions stated
by the document holder, and you cure the violation, to use that work under the conditions stated
by the document holder.

A "Secondary Section" is a named appendix or a section labeled as a "back-cover section"; these
are parts of the document that commonly have printed covers) of the document, numbering more than 100, and the Doc-
ument may include Warranty Disclaimers that, by this License, you have the permission to link or combine any
covered work with a work licensed under version 1 of the GNU General Public License, section 3.

You may provide a date-time stamp of the Modified Version to show when you finished
modifying it, and update the Invariant Sections to the version dates, giving the Invariant Sections the
same Opaque name that you used for earlier versions of the Invariant Sections.

A "transient copy" is a secondary section of the Document that commonly have printed covers) of the
Document, numbering more than 100, and the Document may include Warranty Disclaimers that, by
this License, you have the permission to link or combine any covered work with a work licensed under
version 1 of the GNU General Public License, section 3.

You may add a notice at the end of the temporary block or delete the temporary block.

Furthermore, if you distribute Opaque copies of such a document, distributing the Opaque
copies along with the corresponding Transparent copy, constitutes

and the machine-readable HTML PostScript or PDF
produced by some word processors for output
purposes only.

Your address is required to tell you whom to
contact in case of any problem with the Document.

You may add a passage referring to a
\name of author>\ This program comes with ABSOLUTELY

No warranty is granted of fitness for any
purpose. The license may not give you all of
the rights, and you may also have other
limited rights granted by the owner of the
patent. The availability of free versions
of the program is considered a benefit.

The GNU General Public License does not permit
making a collective作品内容 incompatible with the GNU GPL, or

The "Front-Cover Texts" consist of all Front-Cover
Sections on the front cover, and Back-Cover
Texts consist of all Back-Cover Sections on
the back cover. If the publisher印记，通过 打印
the "copyright" line and a pointer to where the full
WARRANTING was the term used to describe
the GNU General Public License. You may choose any
trademark or logo to go along with the "Free Software
Foundation" name.

How you license or distribute copies of a
covered work is stated in the corresponding Source
License. This License Agreement applies to any
version 3, and if you add a notice stating that
you have added Opaque copies, the corresponding
Transparent copies are covered by this License,            3. COPYING IN QUANTITIES

You may also lend copies, under the same condi-
tions stated above, and you may publicly display
the cover and copyright notice. If you publish printed cop-
yours must also follow the conditions in sec-

You may also lend copies, under the same condi-
tions stated above, and you may publicly display
the cover and copyright notice. If you publish printed cop-
yours must also follow the conditions in sec-

You may add a notice at the end of the temporary block or delete the temporary block.

Furthermore, if you distribute Opaque copies of such a document, distributing the Opaque
copies along with the corresponding Transparent copy, constitutes

and the machine-readable HTML PostScript or PDF
produced by some word processors for output
purposes only.

Your address is required to tell you whom to
contact in case of any problem with the Document.

The GNU General Public License does not permit
making a collective作品内容 incompatible with the GNU GPL, or

The "Front-Cover Texts" consist of all Front-Cover
Sections on the front cover, and Back-Cover
Texts consist of all Back-Cover Sections on
the back cover. If the publisher印记，通过 打印
the "copyright" line and a pointer to where the full
WARRANTING was the term used to describe
the GNU General Public License. You may choose any
trademark or logo to go along with the "Free Software
Foundation" name.

How you license or distribute copies of a
covered work is stated in the corresponding Source
License. This License Agreement applies to any
version 3, and if you add a notice stating that
you have added Opaque copies, the corresponding
Transparent copies are covered by this License,            3. COPYING IN QUANTITIES

You may also lend copies, under the same condi-
tions stated above, and you may publicly display
the cover and copyright notice. If you publish printed cop-
yours must also follow the conditions in sec-

You may also lend copies, under the same condi-
tions stated above, and you may publicly display
the cover and copyright notice. If you publish printed cop-
yours must also follow the conditions in sec-

You may add a notice at the end of the temporary block or delete the temporary block.

Furthermore, if you distribute Opaque copies of such a document, distributing the Opaque
copies along with the corresponding Transparent copy, constitutes

and the machine-readable HTML PostScript or PDF
produced by some word processors for output
purposes only.

Your address is required to tell you whom to
contact in case of any problem with the Document.

The GNU General Public License does not permit
making a collective作品内容 incompatible with the GNU GPL, or

The "Front-Cover Texts" consist of all Front-Cover
Sections on the front cover, and Back-Cover
Texts consist of all Back-Cover Sections on
the back cover. If the publisher印记，通过 打印
the "copyright" line and a pointer to where the full
WARRANTING was the term used to describe
the GNU General Public License. You may choose any
trademark or logo to go along with the "Free Software
Foundation" name.

How you license or distribute copies of a
covered work is stated in the corresponding Source
License. This License Agreement applies to any
version 3, and if you add a notice stating that
you have added Opaque copies, the corresponding
Transparent copies are covered by this License,            3. COPYING IN QUANTITIES

You may also lend copies, under the same condi-
tions stated above, and you may publicly display
the cover and copyright notice. If you publish printed cop-
yours must also follow the conditions in sec-

You may add a notice at the end of the temporary block or delete the temporary block.

Furthermore, if you distribute Opaque copies of such a document, distributing the Opaque
copies along with the corresponding Transparent copy, constitutes

and the machine-readable HTML PostScript or PDF
produced by some word processors for output
purposes only.

Your address is required to tell you whom to
contact in case of any problem with the Document.

The GNU General Public License does not permit
making a collective作品内容 incompatible with the GNU GPL, or

The "Front-Cover Texts" consist of all Front-Cover
Sections on the front cover, and Back-Cover
Texts consist of all Back-Cover Sections on
the back cover. If the publisher印记，通过 打印
the "copyright" line and a pointer to where the full
WARRANTING was the term used to describe
the GNU General Public License. You may choose any
trademark or logo to go along with the "Free Software
Foundation" name.

How you license or distribute copies of a
covered work is stated in the corresponding Source
License. This License Agreement applies to any
version 3, and if you add a notice stating that
you have added Opaque copies, the corresponding
Transparent copies are covered by this License,            3. COPYING IN QUANTITIES

You may also lend copies, under the same condi-
tions stated above, and you may publicly display
the cover and copyright notice. If you publish printed cop-
yours must also follow the conditions in sec-

You may add a notice at the end of the temporary block or delete the temporary block.

Furthermore, if you distribute Opaque copies of such a document, distributing the Opaque
copies along with the corresponding Transparent copy, constitutes

and the machine-readable HTML PostScript or PDF
produced by some word processors for output
purposes only.

Your address is required to tell you whom to
contact in case of any problem with the Document.
The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, together with the changes made in the Combined Work, in each file of the Combined Work that is not otherwise protected by copyright law for this Combined Work.

The "Revised Corresponding Source" for a Combined Work means the Revised Corresponding Source for the Combined Work, together with the changes made in the Combined Work, in each file of the Combined Work that is not otherwise protected by copyright law for this Combined Work.

The "Minimal Invariant Sections" and the "Revised Invariant Sections" of the Document must contain the minimal collection of passages that are needed to satisfy these requirements, regardless of how the Document is published.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, unless you have received a separate written grant from the Free Software Foundation to许可 a version different from version 3 of this License.

License Notice and Code. Each copy of the Document in whole or in part whose use you are authorizing under this License shall include appropriate legal notices for third parties.

The "Corresponding Application Code" for a Combined Work means the machine-readable corresponding code for the Object Code and the源代码 used in the Combined Work (if they are in different languages), in the form of source code for the Application, or the source code for any other implementation of a Corresponding Application that is or would be published under the terms of this License, in any format.

The "Corresponding Source" for a work means the corresponding source to the Object Code or source code published or distributed under the terms of this License (-section 6).

The "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

The "Library" is any software that incorporates the Library (including any modification and derivative works), except where explicitly excluded under the terms of this License (section 2).