| [**Close this window**](about:blank) |
| --- |



| **C# for the High School Classroom** **Introduction**  C# is a modern, object oriented language designed to take advantage of Microsoft’s .NET Framework. It comes from the family of C languages and is the next evolution of C++. Since its public release in February of 2002, C# has grown immensely popular in industry and has also been standardized by ECMA. Not only another evolution of existing technology, C# revolutionizes software development by mixing the power of C++ with the easy and rapid development of languages like Visual Basic and Java. C# also allows for the abstraction of difficult concepts, making it an ideal language for introductory courses in computer science. This guide introduces the fundamental concepts of C# providing necessary and in-depth information to begin immediate development with the .NET Framework.  **Hello, World**  The first program most people write when learning a new language is the inevitable Hello, World. You will get a chance to examine the C# version of this traditional first program. The example code below contains all of the essential elements of a C# program, and it is easy to test!  using System;  class Hello  {  public static void Main()  {  Console.WriteLine("Hello, World");  }  }   | **Note:** For a step-by-step walkthrough of compiling a C# application in Visual Studio .NET, please see [MainFunction’s Introductory Guide to Visual Studio .NET](http://educators.mainfunction.com/articles/%20http:/educators.mainfunction.com/Resources/APCSDownloads/VS%20Getting%20Started.zip). If you are compiling with the .NET Framework SDK command line tools, you can invoke the compiler as shown below.   csc.exe /target:exe /out:HelloWorld.exe /reference:System.dll HelloWorld.cs  For more information on the command line compiler, please visit [Building from the Command Line](http://educators.mainfunction.com/Resources/interchange/Preview.asp?PeerID=358). | | --- |   When executed from the command line, it simply displays the following message:  Hello, World  In C#, an application is a collection of one or more classes, data structures, and other types. A class is defined as a set of data combined with methods (or functions) that can manipulate that data. When you look at the code for the Hello, World application, you will see that there is a single class called Hello. This class is introduced by using the keyword class. Following the class name is an open brace ({). Everything up to the corresponding closing brace (}) is part of the class.  You can spread the classes for a C# application across one or more files. You can put multiple classes in a file, but you cannot span a single class across multiple files.   | **Note for C++ developers:** C# does not distinguish between the definition and the implementation of a class in the same way that C++ does. There is no concept of a definition (.h or .hpp) file. All code for the class is written in one file. | | --- | | **Note for Java developers:** The name of the application file does not need to be the same as the name of the class. |   Every application must start somewhere. When a C# application is run, execution starts at the method called Main. If you are used to programming in C, C++, or even Java, you are already familiar with this concept.   | **Important:** The C# language is case sensitive. Main must be spelled with an uppercase “M” and with the rest of the name in lowercase. | | --- |   Although there can be many classes in a C# application, there can only be one entry point. It is possible to have multiple classes each with Main in the same application, but only one Main will be executed. You need to specify which one should be used when the application is compiled.  The signature of Main is important too. If you use Visual Studio, it will be created automatically as static void. (You will learn what these mean a little later.) Unless you have a good reason, you should not change the signature.   | **Tip:** You can change the signature to some extent, but it must always be static, otherwise it might not be recognized as the application’s entry point by the compiler. | | --- |   The application runs either until the end of Main is reached or until a return statement is executed by Main.  As part of the Microsoft .NET Framework, C# is supplied with many utility classes that perform a range of useful operations. These classes are organized into *namespaces*. (A namespace is a set of related classes.) A C# namespace may also contain other namespaces.  The .NET Framework is made up of many namespaces, the most important of which is called System. The System namespace contains the classes that most applications use for interacting with the operating system. The most commonly used classes handle input and output (I/O). As with many other languages, C# has no I/O capability of its own and therefore depends on the operating system to provide a .NET compatible interface.  You can refer to objects in namespaces by prefixing them explicitly with the identifier of the namespace. For example, the System namespace contains the Console class, which provides several methods, including WriteLine. You can access the WriteLine method of the Console class as follows:  System.Console.WriteLine("Hello, World");  However, using a fully qualified name like in the above example to refer to objects can be unwieldy and error prone. To ease this burden, you can specify a namespace by placing a using directive at the beginning of your application before the first class is defined. A using directive specifies a namespace that will be examined if a class is not explicitly defined in the application. You can put more than one using directive in the source file, but they must all be placed at the beginning of the file.  With the using directive, you can rewrite the previous code as follows:  using System;  ...  Console.WriteLine("Hello, World");  In the Hello, World application, the Console class is not explicitly defined. When the Hello, World application is compiled, the compiler searches for Console and finds it in the System namespace instead. The compiler generates code that refers to the fully qualified name System.Console.   | **Note:** The classes of the System namespace, and the other core functions accessed at run time, reside in an assembly called mscorlib.dll. This assembly is used by default. You can refer to classes in other assemblies, but you will need to specify the locations and names of those assemblies when the application is compiled. | | --- |   **Console I/O**  The Console class provides a C# application with access to the standard input, standard output, and standard error streams. Standard input is normally associated with the keyboard—anything that the user types on the keyboard can be read from the standard input stream. Similarly, the standard output stream is usually directed to the screen, as is the standard error stream.   | **Note:** These streams and the Console class are only meaningful to console applications. These are applications that run in a Command window. | | --- |   You can direct any of the three streams (standard input, standard output, standard error) to a file or device. You can do this programmatically, or the user can do this when running the application.  You can use the Console.Write and Console.WriteLine methods to display information on the console screen. These two methods are very similar; the main difference is that WriteLine appends a new line/carriage return pair to the end of the output, and Write does not.  Both methods are overloaded. You can call them with variable numbers and types of parameters. For example, you can use the following code to print “99” to the screen:  Console.WriteLine(99);  You can use the following code to write the message “Hello, World” to the screen:  Console.WriteLine("Hello, World");  You can use more powerful forms of Write and WriteLine that take a format string and additional parameters. The format string specifies how the data is output, and it can contain markers, which are replaced in order by the parameters that follow. For example, you can use the following code to display the message “The sum of 100 and 130 is 230”:  Console.WriteLine("The sum of {0} and {1} is {2}",  100, 130, 100+130);   | **Important:** The first parameter that follows the format string is referred to as parameter zero: {0}. | | --- |   The format strings can specify additional information about how to display numeric types. After the marker, you can specify the number of characters to be occupied by the formatted number. If this number is less than the required space, the entire number will simply be displayed. If the number is negative, the formatted number will be left justified, otherwise it will be right justified. By following the marker with a colon, you can add a format specifier to explain how a numeric type should be formatted. Some common format specifiers include:   | C | Locale specific currency | | --- | --- | | D | General integer (integers only) | | E | Scientific notation | | F | Fixed point decimal | | G | General number | | N | Locale specific number format | | P | Percentage | | X | Hexadecimal (integers only) |     The following line outputs "Formatted Number: 123,456,789.00" to the console.  Console.WriteLine("Formatted Number:{0,20:N}", 123456789);  Just as the process to output has been simplified, the Console class makes inputting data easier as well. You can obtain user input from the keyboard by using the Console.Read and Console.ReadLine methods.  The Read method reads the next character from the keyboard. It returns the int value –1 if there is no more input available. Otherwise it returns an int representing the character read.  The ReadLine method reads all characters up to the end of the input line (the carriage return character). The input is returned as a string of characters. You can use the following code to read a line of text from the keyboard and display it to the screen:  string input = Console.ReadLine();  Console.WriteLine("{0}", input);  To input numeric types from the console, simply input a line of text and use the Parse method of the native data types. It should be noted that incorrect user input will raise an exception when the data is attempting to be parsed.  Console.WriteLine("What is your age?");  int age = int.Parse(Console.ReadLine());  **Commenting**  C# provides several mechanisms for adding comments to application code: single-line comments, multiple-line comments, and XML-generated documentation.  You can add a single-line comment by using the forward slash characters (//). When you run your application, everything following these two characters until the end of the line is ignored.  // Get the user’s name  Console.WriteLine("What is your name? ");  string name = Console.ReadLine();  You can also use block comments that span multiple lines. A block comment starts with the /\* character pair and continues until a matching \*/ character pair is reached. You cannot nest block comments.  /\* Find the square root of the  following expression \*/  x = (…);  You can use C# comments to generate XML documentation for your applications. Documentation comments begin with three forward slashes (///) followed by an XML documentation tag.  /// <summary> The Hello class prints a greeting  /// on the screen  /// </summary>  class Hello  {  /// <remarks> We use console-based I/O.  /// For more information about WriteLine, see  /// <seealso cref="System.Console.WriteLine"/>  /// </remarks>  public static void Main()  {  Console.WriteLine("Hello, World");  }  }  You can compile the XML tags and documentation into an XML file for further manipulation and presentation. XML commenting falls outside the scope of the average high school classroom, so it will not be covered in further detail here. For more information, visit [Internet Q&A: What is XML?](http://educators.mainfunction.com/Resources/Display.asp?page=qa_xml).  **Variables**  As with any language, it is important to understand the requirements C# places on variable naming, creation, and manipulation.  The following are the naming rules for C# variables:   * Start each variable name with a letter or underscore character. * After the first character, use letters, digits, or the underscore character * Do not use reserved keywords * If you use a disallowed variable name, you will get a compile-time error   Although not required, it is preferred that you follow these additional recommendations when naming your variables:   * Avoid using all uppercase letters * Avoid starting with an underscore * Avoid using abbreviations   **Keywords:** The following is a list of keywords in C#. Remember, you cannot use any of these words as variable names.   | abstract | as | base | bool | break | | --- | --- | --- | --- | --- | | byte | case | catch | char | checked | | class | const | continue | decimal | default | | delegate | do | double | else | enum | | event | explicit | extern | false | finally | | fixed | float | for | foreach | goto | | if | implicit | in | int | interface | | internal | is | lock | long | namespace | | new | null | object | operator | out | | override | params | private | protected | public | | readonly | ref | return | sbyte | sealed | | short | sizeof | stackalloc | static | string | | struct | switch | this | throw | true | | try | typeof | uint | ulong | unchecked | | unsafe | ushort | using | virtual | void | | volatile | while |  |  |  |   C# supports the following native data types (similar to languages like C, C++, and even Java):  sbyte, byte, short, ushort, int, uint, long,  ulong, char, float, double, bool, decimal  Variables that are declared in methods, properties, or indexers are called local variables. Generally, you declare a local variable by specifying the data type followed by the variable name, as shown in the following example:  int itemCount;  You can declare multiple variables in a single declaration by using a comma separator, as shown in the following example:  int itemCount, employeeNumber;  In C#, you cannot declare a variable in an inner block with the same name as a variable in an outer block. For example, the following code is not allowed:  int i;  {  int i; // Error: i already declared in parent block  ...  }  However, you can declare variables with the same name in *sibling* blocks. Sibling blocks are blocks that are enclosed by the same parent block and are nested at the same level. The following is an example:  {  int i;  ...  }  ...  {  int i;  ...  }  You can declare variables anywhere in a statement block. Given this freedom, you can easily follow the recommendation of initializing a variable at the point of declaration.  In C#, you cannot use uninitialized variables. The following code will result in a compile-time error because the loopCount variable has not been assigned an initial value:  int loopCount;  Console.WriteLine("{0}", loopCount);  You use assignment operators to assign a new value to a variable. To assign a value to a variable that is already declared, use the assignment operator (=), as shown in the following example:  int employeeNumber;  employeeNumber = 23;  You can also initialize a variable when you declare it, as shown in the following example:  int employeeNumber = 23;  You can use the assignment operator to assign values to character type variables, as shown in the following example:  char middleInitial = 'J';  **Variable Manipulation**  The following code declares an int variable called itemCount, assigns it the value 2, and then increments it by 40:  int itemCount;  itemCount = 2;  itemCount = itemCount + 40;  The above code to increment a variable works, but it is slightly cumbersome. You need to write the identifier that is being incremented twice. For simple identifiers this is rarely a problem, unless you have many identifiers with very similar names. However, you can use expressions of arbitrary complexity to designate the value being incremented, as in the following example:  items[(index + 1) % 32] = items[(index + 1) % 32] + 40;  In these cases, if you needed to write the same expression twice you could easily introduce a subtle bug. Fortunately, there is a shorthand form that avoids the duplication:  itemCount += 40;  items[(index + 1) % 32] += 40;  This shorthand method works for all arithmetic operators.  var += expression; // var = var + expression  var -= expression; // var = var - expression  var \*= expression; // var = var \* expression  var /= expression; // var = var / expression  var %= expression; // var = var % expression  Incrementing or decrementing a value by one is so common, that the shorthand method has an even shorter form.  itemCount++; // itemCount += 1;  itemCount--; // itemCount -= 1;  The ++ operator is called the increment operator and the –– operator is called the decrement operator. You can think of ++ as an operator that changes a value to its successor and –– as an operator that changes a value to its predecessor. Once again, this shorthand is the preferred idiomatic way for C# programmers to increment or decrement a value by one.  You can use the ++ and –– operators in two forms. You can place the operator symbol *before* the identifier, as shown in the following examples. This is called the *prefix* notation.  ++itemCount;  --itemCount;  You can also place the operator symbol after the identifier, as shown in the following examples. This is called the *postfix* notation.  itemCount++;  itemCount--;  An important feature of C# is that assignment is an operator. This means that besides assigning a value to a variable, an assignment expression itself has a value, or outcome, which is the value of the variable after the assignment has taken place. In most statements the value of the assignment expression is discarded, but it can be used in a larger expression, as in the following example:  int itemCount = 0;  Console.WriteLine(itemCount = 2); // Prints 2  Console.WriteLine(itemCount = itemCount + 40); // Prints 42  Compound assignment is also an assignment. This means that a compound assignment expression, besides assigning a value to a variable, also has a value—an outcome itself. Again, in most statements the value of the compound assignment expression is discarded, but it can be used in a larger expression, as in the following example:  int itemCount = 0;  Console.WriteLine(itemCount += 2); // Prints 2  Console.WriteLine(itemCount -= 2); // Prints 0  Increment and decrement are also assignments. This means, for example, that an increment expression, besides incrementing a variable by one, also has a value, an outcome itself. Again, in most statements the value of the increment expression is discarded, but it can be used again in a larger expression, as in the following example:  int itemCount = 42;  int prefixValue = ++itemCount; // prefixValue == 43  int postfixValue = itemCount++; // postfixValue = 43  The value of the increment expression differs depending on whether you are using the prefix or postfix version. In both cases itemCount is incremented. That is not the issue. The issue is what the value of the increment expression is. The value of a prefix increment/decrement is the value of the variable *after* the increment/decrement takes place. The value of a postfix increment/decrement is the value of the variable *before* the increment/decrement takes place.  When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression x + y \* z is evaluated as x + (y \* z) because the multiplicative operator has higher precedence than the additive operator. For example, an *additive-expression* consists of a sequence of *multiplicative-expressions* separated by + or - operators, thus giving the + and – operators lower precedence than the \*, /, and % operators.  When an operand occurs between two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed. For example, x + y + z is evaluated as (x + y) + z. This is particularly important for assignment operators. For example, x = y = z is evaluated as x = (y = z).   * Except for the assignment operators, all binary operators are *left-associative*, meaning that operations are performed from left to right. * The assignment operators and the conditional operator (? :) are *right-associative*, meaning that operations are performed from right to left.   You can control precedence and associativity by using parentheses. For example, x + y \* z first multiplies y by z and then adds the result to x, but (x + y) \* z first adds x and y and then multiplies the result by z.  [Part 2](http://educators.mainfunction.com/Resources/display.asp?page=Csharp_2) | [Back to the top](http://educators.mainfunction.com/articles/pfv.asp?page=Csharp_1#top#top) | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**Close this window**](about:blank) | | |



| **C# for the High School Classroom Part 2**  **Casting**  In C#, type casting occurs similarly to other languages in the C family. For example:  int intValue = 200;  long longValue = intValue;  Converting from an int data type to a long data type is implicit. This conversion always succeeds, and it never results in a loss of information. Any conversions made from a less precise to a more precise type will succeed.  You can convert variable types explicitly by using a cast expression. Explicit casting is required whenever a loss of precision occurs. The following example shows how to convert the variable longValue from a long data type to an int data type by using a cast expression:  using System;  class Test  {  static void Main( )  {  long longValue = Int64.MaxValue;  int intValue = (int) longValue;  Console.WriteLine("(int) {0} = {1}", longValue, intValue);  }  }  Because an overflow occurs in this example, the output is as follows:  (int) 9223372036854775807 = -1  You can also cast objects similarly. You can cast to any type in a given object’s inheritance hierarchy. One of the most common cases is casting to object, the base class from which all objects in .NET inherit.  string strValue = "Text";  object objValue = (object) strValue;  **Selection Statements**  The if statement is the primary decision-making statement. It can be coupled with an optional else clause, as shown:  if (Boolean-expression)  first-embedded-statement  else  second-embedded-statement  The if statement evaluates a Boolean expression to determine the course of action to follow. If the Boolean expression evaluates to true, the control is transferred to the first embedded statement. If the Boolean expression evaluates to false, and there is an else clause, the control is transferred to the second embedded statement.  You can use a simple embedded if statement such as the following:  if (number % 2 == 0)  Console.WriteLine("even");  Although braces are not required in embedded statements, many style guides recommend using them because they make your code less error prone and easier to maintain. You can rewrite the previous example with braces as follows:  if (number % 2 == 0)  {  Console.WriteLine("even");  }  Implicit conversion from an integer to a Boolean value is a potential source of bugs. To avoid such conversion-related bugs, C# does not support integer to Boolean value conversion. This is a significant difference between C# and other similar languages. For example, the following statements, which at best generate warnings in C and C++, result in compilation errors in C#:  int x;  ...  if (x) ... // Must be x != 0 in C#  if (x = 0) ... // Must be x == 0 in C#  You can handle cascading if statements by using an else if statement. C# does not support the else if statement but forms an else if-type statement from an else clause and an if statement, as in C and C++. Languages such as Visual Basic support cascading if statements by using an else if statement between the initial if statement and the final else statement.  By using the else if construct, you can have any number of branches. However, the statements controlled by a cascading if statement are mutually exclusive, so that only one statement from the set of else if constructs is executed.  The switch statement provides an elegant mechanism for handling complex conditions that would otherwise require nested if statements. It consists of multiple case blocks, each of which specifies a single constant and an associated case label. You cannot group a collection of constants together in a single case label. Each constant must have its own case label.  A switch block can contain declarations. The scope of a local variable or constant that is declared in a switch block extends from its declaration to the end of the switch block, as is shown in the example on the slide.  A switch statement is executed as follows:   1. If one of the constants specified in a case label is equal to the value of the switch expression, control is transferred to the statement list following the matched case label. 2. If no case label constant is equal to the value of the switch expression, and the switch statement contains a default label, control is transferred to the statement list following the default label.<> 3. If no case label constant is equal to the value of the switch expression, and the switch statement does not contain a default label, control is transferred to the end of the switch statement.   You can use a switch statement to evaluate only the following types of expressions: any integer type, a char, an enum, or a string. You can also evaluate other expression types by using the switch statement, as long as there is exactly one user-defined implicit conversion from the disallowed type to one of the allowed types.  **Iteration Statements**  The while statement is the simplest of all iteration statements. It repeatedly executes an embedded statement *while* a Boolean expression is true. Note that the expression that the **while** statement evaluates must be Boolean, since C# does not support implicit conversion from an integer to a Boolean value.  int i = 0;  while (i < 10)  {  Console.WriteLine(i);  i++;  }  A while statement is executed as follows:   1. The Boolean expression controlling the while statement is evaluated. 2. If the Boolean expression yields true, control is transferred to the embedded statement. When control reaches the end of the embedded statement, control is implicitly transferred to the beginning of the while statement, and the Boolean expression is re-evaluated. 3. If the Boolean expression yields false, control is transferred to the end of the while statement. Therefore, while the controlling Boolean expression is true, the program repeatedly executes the embedded statement.   The Boolean expression is tested at the start of the while loop. Therefore, it is possible that the embedded statement may never be executed at all.  A do statement is always coupled with a while statement. It is similar to a while statement, except that the Boolean expression that determines whether to continue or exit the loop is evaluated at the end of the loop rather than at the start. This means that, unlike a while statement, which iterates zero or more times, a do statement iterates one or more times.  Therefore, a do statement always executes its embedded statement at least once. This behavior is particularly useful when you need to validate input before allowing program execution to proceed.  int i = 0;  do  {  Console.WriteLine(i);  i++;  }  while (i < 10);  A do statement is executed as follows:   1. Control is transferred to the embedded statement. 2. When control reaches the end of the embedded statement, the Boolean expression is evaluated. 3. If the Boolean expression yields true, control is transferred to the beginning of the do statement. 4. If the Boolean expression yields false, control is transferred to the end of the do statement.   When using while statements, developers often forget to update the control variable. The following code provides an example of this mistake:  int i = 0;  while (i < 10)  Console.WriteLine(i); // Mistake: no i++  This mistake occurs because the developer’s attention is focused on the body of the while statement and not on the update. Also, the while keyword and the update code may be very far apart.  You can minimize these errors by using the for statement. The for statement overcomes the problem of omitted updates by moving the update code to the beginning of the loop, where it is harder to overlook. The syntax of the for statement is as follows:  for ( *initializer ; condition ; update )*  *embedded-statement*   | **Important:** In a for statement, the update code precedes the embedded statement. Nevertheless, the update code is executed by the runtime after the embedded statement. | | --- |   As with all iteration statements, the condition in a for block must be a Boolean expression that serves as a continuation condition and not a termination condition.  for (int i = 0; i < 10; i++)  {  Console.WriteLine(i);  Console.WriteLine(10 – i);  }  C# also provides the foreach statement, which allows you to iterate through a collection, or structured grouping of elements, without using multiple statements. Rather than explicitly extracting each element from a collection by using syntax specific to the particular collection, you use the foreach statement to approach the problem in the opposite way. You effectively instruct the collection to present its elements one at a time. Instead of taking the embedded statement to the collection, the collection is taken to the embedded statement.  int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  ...  foreach (int i in array)  {  Console.WriteLine(i);  }   | **Important:** The elements of a collection being iterated by a foreach statement are immutable while within the foreach loop. For a more detailed explanation of this restriction, please visit [Scripting .NET using Mondrian](http://educators.mainfunction.com/Resources/interchange/Preview.asp?PeerID=377). | | --- |   **Methods**  Functions in C# are very similar to the C family of languages style. There is one caveat though; C# does not allow global functions. Therefore all functions are technically methods of a certain class.  When creating a method, you must specify the following:   * Name You cannot give a method the same name as a variable, a constant, or any other non-method item declared in the class. The method name can be any allowable C# identifier, and it is case sensitive. * Parameter list The method name is followed by a parameter list for the method. This is enclosed between parentheses. The parentheses must be supplied even if there are no parameters, as is shown in the examples on the slide. * Body of the method Following the parentheses is the body of the method. You must enclose the method body within braces ({ and }), even if there is only one statement.   To create a method, use the following syntax:  access modifiers void MethodName(method parameters)  {  method body  }  The access modifiers indicate various details about how the method can be invoked, overridden, etc., from within and outside the class. The return type of the method is the type of the result the function wished to pass back to the point of invocation. If the method does not with to return a value, the return type should be void, as in the example function. The return statement is not required in a void method, but can be used by itself to exit the method.  After you define a method, you can call it from within the same class and from other classes. To call a method, use the name of the method followed by a parameter list in parentheses. The parentheses are required even if the method that you call has no parameters, as shown in the following example.  MethodName();  To allow methods in one class to call methods in another class, you must declare the method that is called with the public keyword. When calling, you need to specify which class contains the method you want to call using the following syntax:  ClassName.MethodName( );  The following example shows how to call the method TestMethod, which is defined in class A, from Main in class B:  using System;  class A  {  public static void TestMethod()  {  Console.WriteLine("This is TestMethod in class A");  }  }  class B  {  static void Main()  {  A.TestMethod();  }  }  If you do not declare a method as public, it becomes private to the class by default. For example, if you omit the public keyword from the definition of TestMethod, the compiler will display the following error: "‘A.TestMethod()’ is inaccessible due to its protection level."  You can also use the private keyword to specify that the method can only be called from inside the class. The following two lines of code have exactly the same effect because methods are private by default:  private void MyMethod();  void MyMethod();  The public and private keywords shown above specify the *accessibility* of the method. These keywords control whether a method can be called from outside of the class in which it is defined.  You can use the return statement to make a method return immediately to the caller. Without a return statement, execution usually returns to the caller when the last statement in the method is reached.  To declare a method so that it will return a value to the caller, replace the void keyword with the type of the value that you want to return.  The return keyword followed by an expression terminates the method immediately and returns the expression as the return value of the method. The following example shows how to declare a method named TwoPlusTwo that will return a value of 4 to Main when TwoPlusTwo is called:  class ExampleReturningValue  {  static int TwoPlusTwo( )  {  int a,b;  a = 2;  b = 2;  return a + b;  }  static void Main( )  {  int x;  x = TwoPlusTwo( );  Console.WriteLine(x);  }  }  Note that the returned value is an int. This is because int is the return type of the method. When the method is called, the value 4 is returned. In this example, the value is stored in the local variable x in Main.  If you declare a method with a non-void type, you must add at least one return statement. The compiler attempts to check that each non-void method returns a value to the calling method in all circumstances. If the compiler detects that a non-void method has no return statement, it will display the following error message: "Not all code paths return a value." You will also see this error message if the compiler detects that it is possible to execute a non-void method without returning a value.   | **Tip:** You can only use the return statement to return one value from each method call. If you need to return more than one value from a method call, you can use the ref or out parameters, which are discussed later. Alternatively, you can return a reference to an array or class or struct, which can contain multiple values. The general guideline that says to avoid using multiple return statements in a single method applies equally to non-void methods. | | --- |   **Method Parameters**  Parameters allow information to be passed into and out of a method. When you define a method, you can include a list of parameters in parentheses following the method name. In the examples so far in this module, the parameter lists have been empty.  Each parameter has a type and a name. You declare parameters by placing the parameter declarations inside the parentheses that follow the name of the method. The syntax that is used to declare parameters is similar to the syntax that is used to declare local variables, except that you separate each parameter declaration with a comma instead of with a semicolon. The following example shows how to declare a method with parameters:  static void MethodWithParameters(int n, string y)  {  // ...  }  This example declares the MethodWithParameters method with two parameters: n and y. The first parameter is of type int, and the second is of type string. Note that commas separate each parameter in the parameter list.  **The calling code must supply the parameter values when the method is called. The following code shows two examples of how to call a method with parameters. In each case, the values of the parameters are found and placed into the parameters n and y at the start of the execution of MethodWithParameters.**  MethodWithParameters(2, "Hello, world");  int p = 7;  string s = "Test message";  MethodWithParameters(p, s);  In applications, most parameters are used for passing information into a method but not out. Therefore, pass by value is the default mechanism for passing parameters in C#.  The simplest definition of a parameter is a type name followed by a variable name. This is known as a *value parameter*. When the method is called, a new storage location is created for each value parameter, and the values of the corresponding expressions are copied into them.  The expression supplied for each value parameter must be the same type as the declaration of the value parameter, or a type that can be implicitly converted to that type. Within the method, you can write code that changes the value of the parameter. It will have no effect on any variables outside the method call.  In the following example, the variable x inside AddOne is completely separate from the variable k in Main. The variable x can be changed in AddOne, but this has no effect on k.  static void AddOne(int x)  {  x++;  }  static void Main()  {  int k = 6;  AddOne(k);  Console.WriteLine(k); // Display the value 6, not 7  }  A reference parameter is a reference to a memory location. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same location in memory as the variable that is supplied in the method call.  You can declare a reference parameter by using the ref keyword before the type name, as shown in the following example:  static void ShowReference(ref int nId, ref long nCount)  {  // ...  }  The ref keyword only applies to the parameter following it, not to the whole parameter list. Consider the following method, in which nId is passed by reference but longVar is passed by value:  static void OneRefOneVal(ref int nId, long longVar)  {  // ...  }  When calling the method, you supply reference parameters by using the ref keyword followed by a variable. The value supplied in the call to the method must exactly match the type in the method definition, and it must be a variable, not a constant or calculated expression.  int x;  long q;  ShowReference(ref x, ref q);  If you omit the ref keyword, or if you supply a constant or calculated expression, the compiler will reject the call, and you will receive an error message similar to the following: "Cannot convert from ‘int’ to ‘ref int.’"  If you change the value of a reference parameter, the variable supplied by the caller is also changed, because they are both references to the same location in memory. The following example shows how changing the reference parameter also changes the variable:  static void AddOne(ref int x)  {  x++;  }  static void Main( )  {  int k = 6;  AddOne(ref k);  Console.WriteLine(k); // Display the value 7  }  This works because when AddOne is called, its parameter x is set up to refer to the same memory location as the variable k in Main. Therefore, incrementing x will increment k.  A ref parameter must be definitively assigned at the point of call; that is, the compiler must ensure that a value is assigned before the call is made. The following example shows what happens if a reference parameter k is not initialized before its method AddOne is called:  int k;  AddOne(ref k);  Console.WriteLine(k);  The C# compiler will reject this code and display the following error message: "Use of unassigned local variable '*k*.'"  Output parameters are like reference parameters, except that they transfer data out of the method rather than into it. Like a reference parameter, an output parameter is a reference to a storage location supplied by the caller. However, the variable that is supplied for the out parameter does not need to be assigned a value before the call is made, and the method will assume that the parameter has not been initialized on entry. Output parameters are useful when you want to be able to return values from a method by means of a parameter without assigning an initial value to the parameter.  To declare an output parameter, use the keyword out before the type and name, as shown in the following example:  static void OutDemo(out int p)  {  // ...  }  As with the ref keyword, the out keyword only affects one parameter, and each out parameter must be marked separately. When calling a method with an out parameter, place the out keyword before the variable to be passed, as in the following example.  int n;  OutDemo(out n);  In the body of the method being called, no initial assumptions are made about the contents of the output parameter. It is treated just like an unassigned local variable. The out parameter must be assigned a value inside the method.  It is sometimes useful to have a method that can accept a varying number of parameters. In C#, you can use the params keyword to specify a variable length parameter list. When you declare a variable-length parameter, you must:   * Declare only one params parameter per method. * Place the parameter at the end of the parameter list. * Declare the parameter as a single-dimension array type.   The following example shows how to declare a variable-length parameter list:  static long AddList(params long[ ] v)  {  long total;  long i;  for (i = 0, total = 0; i < v.Length; i++)  total += v[i];  return total;  }  Because a params parameter is always an array, all values must be the same type. When you call a method with a variable-length parameter, you can pass values to the params parameter in one of two ways:   * As a comma separated list of elements (the list can be empty) * As an array   The following code shows both techniques. The two techniques are treated in exactly the same way by the compiler.  static void Main( )  {  long x;  x = AddList(63, 21, 84); // List  x = AddList(new long[ ]{ 63, 21, 84 }); // Array  }  Regardless of which method you use to call the method, the params parameter is treated like an array. You can use the Length property of the array to determine how many parameters were passed to each call. In a params parameter, a copy of the data is made, and although you can modify the values inside the method, the values outside the method are unchanged.  Methods cannot have the same name as other non-method items in a class. However, it is possible for two or more methods in a class to share the same name. Name sharing among methods is called overloading. Overloaded methods are methods in a single class that have the same name. The C# compiler distinguishes overloaded methods by comparing the parameter lists. The following code shows how you can use different methods with the same name in one class:  class OverloadingExample  {  static int Add(int a, int b)  {  return a + b;  }  static int Add(int a, int b, int c)  {  return a + b + c;  }  static void Main( )  {  Console.WriteLine(Add(1,2) + Add(1,2,3));  }  }  The signature of a method consists of the name of the method, the number of parameters that the method takes, and the type and modifier (such as out or ref) of each parameter. The following three methods have different signatures, so they can be declared in the same class.  **Arrays**  You use the same notation to declare an array that you would use to declare a simple variable. First, specify the type, and then specify the name of the variable followed by a semicolon. You declare the variable type as an array by using square brackets. Many other programming languages, such as C and C++, also use square brackets to declare an array. Other languages, like Microsoft® Visual Basic®, use parentheses.  In C#, array notation is very similar to the notation used by C and C++, although it differs in two subtle-but-important ways:   * You cannot write square brackets to the right of the name of the variable. * You do not specify the size of the array when declaring an array variable.   The following are examples of allowed and disallowed notation in C#:  type[] name; // Allowed  type name[]; // Not allowed in C#  type[4] name; // Also not allowed in C#  To declare a one-dimensional array variable, you use unadorned square brackets as shown on the slide. Such an array is also called an array of rank 1 because one integer index associates with each element of the array.  To declare a two-dimensional array, you use a single comma inside the square brackets, as shown on the slide. Such an array is called an array of rank 2 because two integer indexes associate with each element of the array. This notation extends in the obvious way: each additional comma between the square brackets increases the rank of the array by one. You do not include the length of the dimensions in the declaration for an array variable.  Array indexes (for all ranks) start from zero. To access the first element inside a row, use the expression:  row[0]    Rather than the expression:  row[1]  Indexing from 0 means that the last element of an array instance containing size elements is found at [size-1] and not at [size]. Accidentally using [size] is a common off-by-one error, especially for programmers used to a language that indexes from one, such as Visual Basic.  In C#, an array element access expression is automatically checked to ensure that the index is valid. This implicit bounds check cannot be turned off. Bounds checking is one of the ways of ensuring that C# is a type-safe language.  The Length property of an array is the total length of the array, regardless of the rank of the array. To determine the length of a specific dimension, you can use the GetLength method, as follows:  for (int r = 0; r < grid.GetLength(0); r++)  {  for (int c = 0; c < grid.GetLength(1); c++)  {  Console.WriteLine(grid[r,c]);  }  }  Declaring an array variable does not actually create an array instance. This is because arrays are reference types and not value types. You use the new keyword to create an array instance, also referred to as an array creation expression. You must specify the size of all rank lengths when creating an array instance.  The C# compiler implicitly initializes each array element to a default value dependent on the array element type: integer array elements are implicitly initialized to 0, floating-point array elements are implicitly initialized to 0.0, and Boolean array elements are implicitly initialized to false.  You can use an array initializer to initialize the values of the array instance elements. An array initializer is a sequence of expressions enclosed by curly braces and separated by commas. Array initializers are executed from left to right and can include method calls and complex expressions, as in the following example:  int[ ] data = new int[4]{1, 2, 3 4};  You can only use this convenient shorthand notation when you initialize an array instance as part of an array variable declaration and not as part of an ordinary assignment statement.  int[] data1 = new int[4]{0, 1, 2, 3}; // Allowed  int[] data2 = {0, 1, 2, 3}; // Allowed  data2 = new int[4]{0, 1, 2, 3}; // Allowed  data2 = {0, 1, 2, 4}; // Not allowed  When initializing arrays, you must explicitly initialize all array elements. It is not possible to let trailing array elements revert back to their default value of zero:  int[] data3 = new int[2]{}; // Not allowed  int[] data4 = new int[2]{42}; // Still not allowed  int[] data5 = new int[2]{42,42}; // Allowed  You must explicitly initialize all array elements regardless of the array dimension:  int[,] data = new int[2,3] { // Allowed  {42, 42, 42},  {42, 42, 42}};  int[,] data = new int[2,3] { // Not allowed  {42, 42},  {42, 42, 42}};  int[,] data = new int[2,3] { // Not allowed  {42},  {42, 42, 42}};  You can create arrays by using run-time expressions for the length of each dimension, as shown in the following code:  int size = int.Parse(System.Console.ReadLine());  int[] data = new int[size];  There is one minor restriction. You cannot use a run-time expression to specify the size of an array in combination with array-initializers:  int size = int.Parse(System.Console.ReadLine());  int[] data = new int[size]{0,1,2,3}; // Not allowed  The System.Array class (a class that all arrays implicitly support) provides many methods that you can use when working with arrays. A few of the commonly used methods will be described. The Sort method performs an in-place sort on an array provided as an argument. You can use this method to sort arrays of structures and classes as long as they support the IComparable interface.  int[] data = {4,6,3,8,9,3}; // Unsorted  System.Array.Sort(data); // Now sorted  The Clear method resets a range of array elements to zero (for value types) or null (for reference types), as shown:  int[] data = {4,6,3,8,9,3};  System.Array.Clear(data, 0, data.Length);  The Clone method creates a new array instance whose elements are copies of the elements of the cloned array. You can use this method to clone arrays of user-defined structs and classes. Following is an example:  int[] data = {4,6,3,8,9,3};  int[] clone = (int [])data.Clone();   | **Caution:** The Clone method performs a shallow copy. If the array being copied contains references to objects, the references will be copied and not the objects; both arrays will refer to the same objects. | | --- |   [Part 1](http://educators.mainfunction.com/Resources/display.asp?page=Csharp_1) | [Part 3](http://educators.mainfunction.com/Resources/display.asp?page=Csharp_3) | [Back to the top](http://educators.mainfunction.com/articles/pfv.asp?page=Csharp_2#top#top) | | |
| --- | --- | --- | --- | --- | --- | --- |
| [**Close this window**](about:blank) | | |



| **C# for the High School Classroom Part 3** **Collections**  The System.Collections namespace contains interfaces and classes that define various collections of objects, such as lists, queues, arrays, hashtables, and dictionaries.  IList is an interface for classes that represent an ordered collection of objects that can be individually indexed. Array, ArrayList, StringCollection, and TreeNodeCollection are some of the classes that implement IList. The following tables describe the public instance properties that the IList interface implements.   | **Property** | **Description** | | --- | --- | | IsFixedSize | When implemented by a class, gets a value indicating whether the IList has a fixed size. | | IsReadOnly | When implemented by a class, gets a value indicating whether the IList is read-only. | | Item | When implemented by a class, gets or sets the element at the specified index. In C#, this property is the indexer for the IList class. |   The following table describes the public instance methods that the IList interface defines.   | **Method** | **Description** | | --- | --- | | Add | When implemented by a class, adds an item to the IList. | | Clear | When implemented by a class, removes all items from the IList. | | Contains | When implemented by a class, determines whether the IList contains a specific value. | | IndexOf | When implemented by a class, determines the index of a specific item in the IList. | | Insert | When implemented by a class, inserts an item to the IList at the specified position. | | Remove | When implemented by a class, removes the first occurrence of a specific object from the IList. | | RemoveAt | When implemented by a class, removes the IList item at the specified index. |   In the demonstration below, ArrayList implements the IList interface by using an array whose size is dynamically increased as required.  using System;  using System.Collections;    class listSample  {  public static void Main(string[] args)  {  ArrayList fruit = new ArrayList();  fruit.Add("Apple");  fruit.Add("Pear");  fruit.Add("Orange");  fruit.Add("Banana");  Console.WriteLine ("\nList Contains:");  foreach (string item in fruit)  {  Console.WriteLine(item);  }  }  }  The preceding code example displays the following output to the console:  List Contains:  Apple  Pear  Orange  Banana   | **Note:** C# does an excellent job of abstracting the technicalities of reference and value types from developers. For example, although the ArrayList.Add method takes a parameter of type object (which means it must be a reference type declared on the heap), you are allowed to pass it an int (which is a value type declared on the stack). C# performs a process referred to as boxing, which basically wraps a value type in a reference type to avoid problems like this. It similarly will unbox a value type wrapped in a reference type when necessary. For more information on boxing and unboxing, please visit [A Programmer's Introduction to C#](http://educators.mainfunction.com/Resources/interchange/Preview.asp?PeerID=358). | | --- |   IDictionary is an interface for collections of associated keys and values. Each association must have a unique non-null key, but the value of an association can be any object reference, including a null reference.  Hashtable, DictionaryBase, and SortedList are examples of collection classes that implement IDictionary. The IDictionary interface allows the contained keys and values in the items in a collection to be enumerated, but it does not imply any particular sort order. The IDictionaryEnumerator interface inherits from IEnumerator and adds members to return the object’s Key and Value fields individually or in a DictionaryEntry structure.  The following table describes some of the public instance properties that the IDictionary interface implements.   | **Property** | **Description** | | --- | --- | | IsFixedSize | When implemented by a class, gets a value indicating whether the IDictionary has a fixed size. | | IsReadOnly | When implemented by a class, gets a value indicating whether the IDictionary is read-only. | | Item | When implemented by a class, gets or sets the element at the specified index. In C#, this property is the indexer for the class. | | Keys | When implemented by a class, gets an ICollection containing the keys of the IDictionary. | | Values | When implemented by a class, gets an ICollection containing the values in the IDictionary. |   The following table describes some of the public instance methods that the IDictionary interface implements.   | **Method** | **Description** | | --- | --- | | Add | When implemented by a class, adds an entry with the provided key and value to the IDictionary. | | Clear | When implemented by a class, removes all elements from the IDictionary. | | Contains | When implemented by a class, determines whether the IDictionary contains an entry with the specified key. | | GetEnumerator | When implemented by a class, returns an IDictionaryEnumerator for the IDictionary. | | Remove | When implemented by a class, removes the entry with the specified key from the IDictionary. |   The Hashtable class represents a collection of associated keys and values that are organized on the basis of the keys’ hash code. The objects that are used as keys in a Hashtable must implement or inherit the Object.GetHashCode and Object.Equals methods. If key equality were simply reference equality, the inherited implementation of these methods would suffice.  Furthermore, these methods must produce the same results when they are called with the same parameters while the key exists in the Hashtable. Key objects must be immutable as long as they are used as keys in the Hashtable.  The following code creates a hash table of employee ID numbers and names, which is then searched.  using System;  using System.Collections;    class HashTableSample  {  public static void Main(String[] args)  {  //create hash table of employee numbers and names  Hashtable table = new Hashtable();  table.Add("0123","Jay");  table.Add("0569","Brad");  table.Add("1254","Brian");  table.Add("6839","Seth");  table.Add("3948","Rajesh");  table.Add("1930","Lakshan");  table.Add("9341","Kristian");  //now we'll look to see if an employee is in the table  Console.Write("Employee ID: ");  string input = Console.ReadLine();  if (table.Contains(input))  {  Console.WriteLine("Employee {0}: {1}", input, table[input]);  }  else  {  Console.WriteLine("Employee {0} not found.", input);  }  }  }  **Strings**  The string type represents a string of Unicode characters; string is an alias for System.String in the .NET Framework. Although string is a reference type, the equality (==) operator and the inequality (!=) operator are defined to compare the values of string objects, not the references. Comparing the values of string objects makes testing for string equality more intuitive, as in the following example:  string a = "hello";  string b = "hello";  Console.WriteLine( a == b ); // output: True -- same value    The + operator concatenates strings, as in the following example:  string a = "good " + "morning";    The [ ] operator accesses individual characters of a string, as in the following example:  char x = "test"[2]; // x = 's';    String literals are of type string and can be written in two forms: quoted and @-quoted. Quoted string literals, or verbatim strings, are enclosed in quotation marks ("), as in the following example:  "good morning" // a string literal    Quoted string literals can also contain any character literal, including escape sequences, as in the following example:  string a = "\\\u0066\n"; // backslash, letter f, new line    @-quoted string literals start with @ and are enclosed in quotation marks, as in the following example:  @"good morning" // a string literal    The advantage of using @-quoted string literals is that escape sequences are not processed. This makes it easy to write a fully qualified file name, as in the following example:  @"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"    To include a quoted phrase in an @-quoted string, use two pairs of double quotation marks, as in the following example:  @"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.  **Enumerations**  Enumerations are user defined types that function as limited range integers. They are useful when a variable can only have a specific set of values you choose to define.  To declare an enumeration, use the enum keyword followed by the enum variable name and initial values. For example, the following enumeration defines three integer constants, called enumerator values.  enum Color { Red, Green, Blue }  By default, enumerator values start from 0. In the preceding example, Red has a value of 0, Green has a value of 1, and Blue has a value of 2. You can initialize an enumeration by specifying integer literals.  You can declare a variable colorPalette of type Color using the following syntax:  Color colorPalette; // Declare the variable  colorPalette = Color.Red; // Set value  - or -  colorPalette = (Color)0; // Type casting int to Color  To display an enumeration value in readable format, use the following statement:  Console.WriteLine("{0}", colorPalette);  **Structures**  You can use structures to create objects that behave like built-in value types. Because structs are stored inline and are not heap allocated, there is less garbage collection pressure on the system than there is with classes. In the .NET Framework, simple data types such as int, float, and double are all built-in structures.  You can use a structure to group together several arbitrary types, as shown in the following example:  public struct Employee  {  public string firstName;  public int age;  }  This code defines a new type called Employee that consists of two elements: first name and age. To access elements inside the struct, use the following syntax:  Employee companyEmployee; // Declare variable  companyEmployee.firstName = "Joe"; // Set value  companyEmployee.age = 23;  **Classes**  Although classes and structs are semantically different, they do have syntactic similarity. To define a class rather than a struct:  Use the keyword class instead of struct.  Declare your data inside the class exactly as you would for a struct.  Declare your methods inside the class.  Add access modifiers to the declarations of your data and methods.  The simplest two access modifiers are public and private. The meaning of public is “access not limited.” The meaning of private is “access limited to the containing type.” If you do not specify an access modifier when declaring a class member, it will default to private.  To create instances of a class, you must create a reference variable and the corresponding object using the new keyword as demonstrated below.  object o;  o.ToString(); // Will fail since 'o' not instantiated  object o = new object();  o.ToString(); // Will execute  The this keyword implicitly refers to the object that is making an object method call. Static methods cannot use this as they are not called by using an object.  A constructor turns the memory allocated by new into an object. There are two types of constructors: instance constructors and static constructors. Instance constructors are constructors that initialize objects. Static constructors are constructors that initialize classes.  When you create an object, the C# compiler provides a default constructor if you do not write one yourself. Constructors have the same name as the class, no return type, optional arguments (the default constructor has no arguments), and all fields initialized to zero.  The ability to initialize an object in different ways was one of the primary motivations for allowing overloading. Constructors are special kinds of methods, and they can be overloaded exactly like methods. This means you can define different ways to initialize an object. The following code provides an example:  class Overload  {  private int data;  public Overload( ) { this.data = -1; }  public Overload(int x) { this.data = x; }  }  class Use  {  static void Main( )  {  Overload o1 = new Overload( );  Overload o2 = new Overload(42);  ...  }  }   | **Note:** While C# does allow destructors with the familiar ~class-name()  syntax, they behave differently than expected. If you wish to implement destructors,  please visit [A Programmer's Introduction to C#](http://educators.mainfunction.com/Resources/interchange/Preview.asp?PeerID=358) for more information. | | --- |   A property is a class member that provides access to a field of an object. You use a property to associate actions with the reading and writing of an object’s attribute. A property declaration consists of a type and a name and has either one or two pieces of code referred to as accessors. C# includes both get and set accessors. Accessors have no parameters. A property does not need to have both a get accessor and a set accessor. For example, a read-a only property will provide only a get accessor while a write only property provides only a set accessor. The syntax to declare a property is shown below.  class Button  {  private string caption; // Field  public string Caption // Property  {  get  {  return caption;  }  set  {  caption = value;  }  }  }  An indexer is a member that enables an object to be indexed in the same way as an array. Whereas you can use properties to enable field-like access to the data in your class, you can use indexers to enable array-like access to the members of your class. The following code shows how to implement an indexer that provides access to a private array of strings called list:  class StringList  {  private string[ ] list;  public string this[int index]  {  get  {  return list[index];  }  set  {  list[index] = value;  }  }  ...  // Other code and constructors to initialize list  }  Indexers provide a convenient and elegant method for accessing sub items of an object with a familiar array-like notation.  **Inheritance**  To specify that one class is derived from another, you use the following syntax:  class Derived : Base  {  ...  }  When you declare a derived class, the base class is specified after a colon. The white space around the colon is not significant.  A derived class inherits everything from its base class except for the base class constructors and destructors. Public members of the base class are implicitly public members of the derived class. Private members of the base class, though inherited by the derived class, are accessible only to the members of the base class.  Members of a derived class can access all of the protected members of their base class. To a derived class, the protected keyword behaves like the public keyword. However, between two classes that are not related by a derived-class and base-class relationship, protected members of one class act like private members for the other class.  A virtual method specifies an implementation of a method that can be polymorphically overridden in a derived class. Conversely, a non-virtual method specifies the only implementation of a method. You cannot polymorphically override a non-virtual method in a derived class. To declare a virtual method, you use the virtual keyword after the access modifiers.  An overridden method specifies another implementation of a virtual method. You define virtual methods in a base class, and they can be polymorphically overridden in a derived class.  class Token  {  ...  public virtual string Name( ) { ... }  }  class CommentToken : Token  {  ...  public override string Name( ) { ... }  }  You can only use an override method to override an identical inherited virtual method from the base class. An override declaration must be identical in every way to the virtual method it overrides. They must have the same access level, the same return type, the same name, and the same parameters. Additionally, an overridden method is implicitly virtual, so you can override it as well.  You can hide an identical inherited method by introducing a new method into the class hierarchy. The old method that was inherited by the derived class from the base class is then replaced by a completely different method. You use the new keyword to hide a method. The syntax for this keyword is as follows:  class Token  {  ...  public int LineNumber( ) { ... }  }  class CommentToken: Token  {  ...  new public int LineNumber( ) { ... }  }  The new modifier is necessary only when a derived class method hides a visible base class method that has an identical signature.  To prevent derivation from a class that is not designed to act as a base class, C# allows a class to be declared sealed. You cannot derive from a sealed class. The syntax is demonstrated below.  namespace System  {  public sealed class String  {  ...  }  }  **Interfaces**  An interface specifies a syntactic and semantic contract that all derived classes must adhere to. Specifically, an interface describes the *what* part of the contract and the classes that implement the interface describe the *how* part of the contract.  An interface resembles a class without any code. You declare an interface in a similar manner to the way in which you declare a class. To declare an interface in C#, you use the keyword interface instead of class, as demonstrated below.  interface IToken  {  int LineNumber();  string Name();  }  It is recommended that all interface names be prefixed with a capital "I." For example, use IToken rather than Token. The methods declared in an interface are implicitly public. Therefore, explicit public access modifiers are not allowed.  The methods declared in an interface are not allowed to contain method bodies. Neither are the interface property declarations, which are declarations of properties with no body, interface event declarations, which are declarations of events with no body, and interface indexer declarations, which are declarations of indexers with no body.  Although C# permits only single inheritance, it allows you to implement multiple interfaces in a single class. A class must implement all methods of any interfaces it extends, regardless of whether the interfaces are inherited directly or indirectly. A method implemented from an interface must retain the original signature.   | **Note:** C# also allows the use of the abstract keyword to create classes without an explicit implementation, not too dissimilar to interfaces. For more information on abstract classes, visit [Programming in the .NET Environment](http://educators.mainfunction.com/Resources/interchange/Preview.asp?PeerID=387). | | --- |   **Namespaces**  In C#, you can use namespaces to resolve name clashes. C# namespaces are similar to C++ namespaces and Java packages. Internal access is not dependent on namespaces.  You can use namespaces to show the logical structure of classes in a way that can be interpreted by the compiler. You need to specify the structure explicitly in the grammar of the language by using namespaces. For example, instead of writing  public class VendorAWidget { ... }  You would write:  namespace VendorA  {  public class Widget { ... }  }  When you create a class that is located inside a namespace, you must use its fully qualified name if you want to use that class outside its namespace. The fully qualified name of a class includes the name of its namespace. In the example, the class Widget is embedded inside the VendorA namespace. This means that you cannot use the unqualified name Widget outside the VendorA namespace. For example, the following code will not compile if you place it outside the VendorA namespace.  Widget w = new Widget();  You can fix this code by using the fully qualified name for the Widget class, as follows:  VendorA.Widget w = new VendorA.Widget();  As you can see, using fully qualified names makes code long and difficult to read. You can bring class names back into scope with *using-directives*. You can simply declare an outside namespace to be included by the compiler, as show below.  using VendorA;  class Application  {  static void Main()  {  Widget w = new Widget();  }  }  With namespace directives, you can use classes outside their namespaces without using their fully qualified names. In other words, you can make long names short again.  **Exceptions**  Exceptions have become the standard form of error-handling in modern object oriented languages. All C# exceptions derive from the class named Exception, which is a part of the Common Language Runtime. The hierarchy between these exceptions is displayed on the slide. The exception classes provide the following benefits.  Error messages are no longer represented by integer values or enums. The programmatic integer values such as -3 disappear. In their place, you use specific exception classes such as OutOfMemoryException. Each exception class can reside inside its own source file and is decoupled from all other exception classes.  Meaningful error messages are generated. Each exception class is descriptive, clearly and obviously representing a specific error. Instead of a –3, you use a class called OutOfMemoryException. Each exception class can also contain information specific to itself. For example, a FileNotFoundException class could contain the name of the file that was not found.  Object orientation offers a structured solution to error-handling problems in the form of try and catch blocks. The idea is to physically separate the core program statements that handle the normal flow of control from the error-handling statements. Therefore, the sections of code that might throw exceptions are placed in a try block, and the code for handling exceptions in the try block is placed in a separate catch block.  The syntax of a try-catch block is as follows:  try  {  ... Program Logic ...  }  catch (class-type identifier)  {  ... Error-handling Code ...  }  The class type must be System.Exception or a type derived from System.Exception. You are allowed multiple catch blocks, so long as inherited exception classes always precede their parent exceptions. The example below shows three different exceptions being caught.  try  {  Console.WriteLine("First number:");  int i = int.Parse(Console.ReadLine());  Console.WriteLine("Second number: ");  int j = int.Parse(Console.ReadLine());  int k = i / j;  }  catch (OverflowException caught)  {  Console.WriteLine(caught);  }  catch (DivideByZeroException caught)  {  Console.WriteLine(caught);  }  catch (Exception caught)  {  Console.WriteLine(caught);  }  It is worth noting that the last catch block can be left without a parameter and will catch all left over exceptions, just as the Exception class does in the above example.  You can use the throw statement to raise your own exceptions, as shown in the following example:  if (minute < 1 || minute >= 60)  {  string fault = minute + "is not a valid minute";  throw new InvalidTimeException(fault);  // !!Not reached!!  }  In this example, the throw statement is used to raise a user-defined exception, InvalidTimeException, if the time being parsed does not constitute a valid time. Exceptions typically expect a meaningful message string as a parameter when they are created. This message can be displayed or logged when the exception is caught. It is also good practice to throw an appropriate class of exception.  You can only throw an object if the type of that object is directly or indirectly derived from System.Exception. This is different from C++, in which objects of any type can be thrown. You can use a throw statement in a catch block to rethrow the current exception or a new exception.  C# provides the finally clause to enclose a set of statements that need to be executed regardless of the course of control flow. Therefore, if control leaves a try block as a result of normal execution because the control flow reaches the end of the try block, the statements of the finally block are executed. Also, if control leaves a try block as a result of a throw statement or a jump statement such as break, continue, or goto, the statements of the finally block are executed.  Monitor.Enter(x)  try  {  ... Program Logic ...  }  finally  {  // Regardless of what error occurs, the resource  // 'x' needs to be released!  Monitor.Exit(x);  }  The finally block is useful in two situations: to avoid duplication of statements in multiple catch blocks and to release resources after an exception has been thrown like in the example above.  **Conclusion**  Hopefully by now you have been exposed to enough of the C# language to complete most introductory computer science assignments. It is very important to notice how C# simplifies difficult concepts and allows for rapid, yet powerful development.  There are additional advanced aspects of C# that fall outside the realm of most classrooms including delegates, events, operator overloading, attributes, reflection, etc., which have not been covered here for that reason. If you are interested in pursuing further topics in C#, please visit [Programming in the .NET Environment](http://educators.mainfunction.com/Resources/interchange/Preview.asp?PeerID=387) for a comprehensive reference.  If you are interested in C#, .NET, or other Microsoft development, be sure to keep checking in with MainFunction for the latest high school resources.  [Part 1](http://educators.mainfunction.com/Resources/display.asp?page=Csharp_1) | [Part 2](http://educators.mainfunction.com/Resources/display.asp?page=Csharp_2) | [Back to the top](http://educators.mainfunction.com/articles/pfv.asp?page=Csharp_3#top#top) |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |