

Forum:[Hack Forums](#)

Thread: [Welcome to C# | A Comprehensive Guide](#)

Odus - February 19, 2016, 12:22 pm

Hello and welcome to the monster guide to all things C#! This thread aims to teach you the fundamentals of programming and give you some practice. There are also some more advanced concepts covered here as well. We will start with what's known as a console application (think command prompt on Windows).

This guide assumes no previous knowledge, but if you're not a beginner and feel I've missed or gotten something incorrect then please let me know. Ready to get started?

Table of Contents IntroductionWhat is C# and .NET?

The IDE

Creating a Project

Getting Started

Hello World!

[Fundamentals](#)Data Types

Variables

Converting

Naming

Operators

Commentary

Syntax Highlighting

[Beginner](#)Boolean Logic

Conditionals

Switches

Iteration

Collections

Methods

[Intermediate](#)Globals

Classes

Enumerations

Scope

Properties

Ternaries

Parameters

[Advanced](#)Parameters II (Out)

String Interpolation

Inheritance

Overloads

Overrides

Collections II (Generics)

File Input/Output

Pointers
PInvoke
Threading

Introduction What is C# and .NET? It's important we take some time to learn about the language itself, we are going to be using it after all. The C# syntax (format) was modeled around the C family, but took some inspiration from Java to give you the best of both worlds.

C# as a language isn't just limited to desktop development, you can create many things with the language. For example, Xamarin, a mobile development platform, allows you to write your apps in C# and deploy them across all 3 mobile platforms. There's also a powerful engine named Unity which can create impressive games.

We will be working with the .NET framework, one of Microsoft's best creations developed over the course of around 14 years. In a nutshell, it's a nutshell of programming libraries which seriously accelerates the process of developing software. In other languages such as C or C++, you must handle considerably more elements of your software, such as memory management and writing basic things such as outputting a file yourself.

The .NET framework aims to make life that bit easier by sacrificing trace levels of performance for overall quality of life, and it's well worth it. The world appears to agree as .NET employment - specifically C# - is on the rise.

The IDE 'IDE' stands for Integrated Development Environment. Think of it as software that creates software. A basic IDE will include a code editor for writing your code, a compiler to translate it into (almost) machine-readable code in .NET's case, and a debugger for your application.

Our IDE of choice is Visual Studio which is potentially the best IDE ever created. It's packed with features to make your life as a programmer that bit easier, and it's such a pleasant environment to work in. There are tools such as IntelliSense which mean you don't have to write out your code fully, it will make a guess and refine it as you continue writing.

There are two primary editions of Visual Studio: [Community](#) and [Enterprise](#). The former is free and as a beginner will more than serve your needs. The latter, as per the name, targets industrial/professional development and is expensive. But if you're a student, it's your lucky day since student licenses are free. I'm using Enterprise for this tutorial.

Throughout your time, you may see people still using VS predecessors such as 2010 edition. These people claim it's more lightweight and kinder to less powerful computing systems, but even Enterprise can be toned down to match your system specs. Don't be fooled.

Installing may take some time, especially if you don't have any of the prerequisites or frameworks already installed. Return when you're ready.

Creating a Project When Visual Studio completes, it will look something like this:
(picture from an other tutorial of mine) [\(Click to View\)](#)

Before we can actually do anything, we need to create a project first so fire right in and press New Project (Ctrl + Shift + N).

You will be presented with a window that looks like this:

As I mentioned before, we will be starting with console applications, so select the third option and

name your project "CSConsole" and create. After it's done, you will be presented with the code editor: [\(Click to View\)](#)

This is what we can see: Left- Toolbox Holds our controls (buttons, text boxes, etc.) and would be populated if we were working with WinForms, but more on that later so ignore the window for now. It can also hold templates of code we've already written.

Top- Toolbar Contains our comment & bookmark controls, project states and debug shuttle buttons. As of now, all we care about is the big one that says 'Start'.

Upper Right- Solution Explorer Holds all elements of our project. We can access the properties, view resources and navigate between code files here. Again, not necessary as of now.

Lower Right- Properties This property view will allow us to configure aspects of a control at design time. For example, we can set the name or placeholder of a TextBox there.

Bottom- Output & Errors When building or debugging, the output window acts as our own little console. If there is no error tab, press the View tab at the top and select 'Error List'. It displays information for any compile or build errors and their descriptions & locations.

Getting Started Now it's time to dissect this code that Visual Studio has created for us. I know it looks daunting, but don't worry. Breaking it down will make it easier.

At the very top is:

```
Code using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

This set of instructions must be right at the top of our project. It states we wish to use members of each namespace. This is known as the using directive. Although it's not necessary, it saves us having to qualify it each time we access something in it. For example, if I were writing to the console like so:

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

The using System statement would brighten up and show us that we've accessed something in it. If we removed it, we would get a context error. It's also possible to access by writing:

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

But shortening it is the best route if you're going to use a namespace more than once.

Next, we see:

```
Code namespace CSConsole
```

A namespace is a holder for multiple objects. This can be classes, structs, enums and whatever else. We just studied the System namespace above, so continuing with this example, it would look like:

```
Code namespace System
```

```
{
    public class Console
    {
        public static void WriteLine(string value)
        {

        }
    }
}
```

```
public class Environment
```

```
{  
  
}  
}
```

Inside System is a class named Console, and it has a method (sub procedure) named WriteLine where we can supply a string for it to output to the console. You can also see there is another class there named Environment which is there to demonstrate namespaces can hold multiple classes.

Don't worry if you don't understand the exact details, what's important is that you understand this hierarchy model and how various elements are grouped together.

Finally, we have this line:

```
Code static void Main(string[] args)
```

The declaration of what's called a method. Inside these brackets is where the actual code we write will be run. Ignore static for now, the main parts of this are: void Specifies that this particular method does not return anything. It's called to run a sequence of instructions and that's it.

Main The name of our method. In this case, it's named 'Main'. This method is known as the entry point of an application, it's the first method that will be called when started.

(string[] args) Inside the parenthesis is the opportunity to pass arguments (args) to the method which will be used somehow when it's called. In this case, our args is a string array, but don't worry about that. Think of when you drag a file over a program's icon and it starts using that file. Windows has supplied the location of that file to the program's start arguments.

You'll notice that between all these declarations are curly braces. Each of these marks the beginning and end of the declaration. So everything inside the namespace braces is part of the namespace. Everything inside the class (which is contained in the namespace) is part of the class. This all applies even with expressions such as conditionals and loops which you'll learn about later. Again, this is the hierarchical chain I'm trying to show you.

Hello World! As is tradition, we're going to start simply by writing out "Hello World!" to the console. This practice has been used since the dawn of modern programming to test how complex a language is by simply printing something out and to test compilation.

We've actually already covered this in the namespace section, but once more, this is the line we'll use:

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

This line is accessing the Console class (part of the System namespace, remember?) and calling the WriteLine void (returns nothing, just carries out a set of instructions). Remember I said between a method is what will be executed? Well we are going to place this line inside the Main method.

Now, press Start and it will build, then run. But hopefully you noticed something... It printed then closed. This is exactly what we want. WriteLine was our only instruction. When there is nothing left to execute and nothing to await, it will pack up and close automatically.

From experience, you should know it's a two way stream since the console accepts input as well as output. So if we await input, it gives us time to read what happened. Therefore, we'll call a method from Console named 'ReadLine'.

Can you guess how that might look? ([Click to View](#))

```
Code Console.ReadLine();
```

Place it below the WriteLine, since instructions are effectively read sequentially i.e. line-by-line. Build and run again, you'll see it worked and we can press Enter to terminate.

Fundamentals Data Types Throughout programming, we will be working with data and need something to hold it with. This is known as a variable, which stores one value and can change, but its name and type remain constant. The type of a variable reflects what kind of data the variable will hold and each type comes with certain features and functions, known as members. Here are the core/primitive data types in C#:

- byte Holds an unsigned 8-bit integer 0..255
- sbyte A signed byte, capable of storing negatives -128..127
- short Signed 16-bit integer -32,768..32,767
- ushort Unsigned 16-bit integer 0..65,535
- int A signed 32-bit integer, the most commonly used numeric data type.
-2,147,483,648..2,147,483,647
- uint Unsigned 32-bit integer 0..4,294,967,295
- long A signed 64-bit integer, capable of storing larger values
-9,223,372,036,854,775,808..9,223,372,036,854,775,807
- ulong Storing larger values like the 64-bit integer long, but unsigned
0..18,446,744,073,709,551,615
- float Stores 32-bit floating-point values at 7-digit precision -3.402823e38..3.402823e38
- double As above but this time 64-bit at 15-16-digit precision
-1.79769313486232e308..1.79769313486232e308
- decimal Lower range but higher precision at 128-bit (think monetary)
-79228162514264337593543950335..79228162514264337593543950335
- char Stores one single unicode character Can be literal, hex, cast, or unicode
- bool A two-state logic variable Represents either true or false

There are two others I haven't included as primitive, they are string and object. The first isn't considered fully primitive. Java represents strings as a class and languages like C which is instead an array of type char that's listed above. The object type is a base class which types will inherit from, in some way.

Nevertheless, they are declared as just the same.

You'll also notice the words 'signed' and 'unsigned' littered across our type table. I left this until last in case you figured it out, but signed is able to represent negative values and unsigned isn't.

This is down to very low-level stuff (as in closer to hardware, not in terms of ease). But one bit is usually reserved for the state flag (positive or negative). There's also two's complement storage which you can look up if you feel compelled, but I don't want to bore you or confuse you so we'll move on!

Variables Now that you have a map of core variables, it's time to put this into practice. But first, below is a test showing you various values and it's your goal to guess their type.

"Hello world!"

-12345678

1234.56

"true"

true

't'

3

Answers [\(Click to View\)](#)

string - Remember our Hello World line?

int - Negative and out of type short's range

double - Decimal points

string - Valid as bool but wrapped in quotes, therefore string
bool - Either true or false
char - The one-character length is valid as a string but chars are identified by inverted commas rather than speech marks
byte - Acceptable as an int, short, etc. but can go even lower (8-bit), therefore byte

Using these in C# is very simple. This is how they're declared:

Code `<type> <name> = <value>;`

`int x = 1;`

As you can see, it's very concise and easy to understand. The only thing there we haven't covered is the semi colon (;) at the end, which represents the end of the line. Breaking that statement down:

`<type>`The type of data our variable will hold (int, string, byte, etc.)

`<name>`The alias for our variable which we will write when referencing. This can be named anything you like, it's named 'x' in the example

=Known as an operator, this specifies that the variable will be initialized with the value on the right

`<value>`An initial value for the variable, it's optional but recommended. If you don't include this, don't include an assignment operator either

C# is a strong-typed language, meaning that variable types must be explicit. Unlike languages such as Javascript where you can freely twist and shape a variable as you please, something like this wouldn't work:

Code `int x = 5;`

`x = "hello";`

You would see the code editor render it as a type conversion error under the string, underlined with a squiggly red line. That isn't the only reason, certain types have certain members associated with them. For example, strings have a property named Length (myString.Length) which represents the number of characters in the string (remember, string is essentially a collection of type char). Such a property wouldn't be available in a type like an int.

What makes a program is being able to work with data you don't know, but we need to get this data first. The ReadLine() method we called to suspend the console will read the user input into a string. Like so:

Code `string input = Console.ReadLine();`

We will create a program to welcome a user by his name which we prompt for. To do this, we must:
Prompt the user to enter their name

Read their name into memory (a variable)

Print out their name with a welcome message

Write out the a line saying "Enter your name:" and then read their input into a variable of type string named "name".

Attempt the the first and second parts yourself, collapse the spoiler if you can't get it. ([Click to View](#))

Code `Console.WriteLine("Enter your name:");`

`string name = Console.ReadLine();`

The next part is to join the welcome string with their name. We will look at constants for this. Constants are similar to variables, but they can only be initialized once, the compiler will not permit an attempt to change it. It looks like this:

Code `const string welcomeMsg = "Welcome, ";`

Building strings in programming is known as string concatenation. It's done like this:

Code `Console.WriteLine(welcomeMsg + name);`

Which should produce on the console:

Quote:Welcome, Odus

To get some more practice using variables, we will create a program that calculates the perimeter of a rectangle. Declare an integer named 'width' and call `ReadLine()` to get the user's input.

You'll get an error. The next chapter goes on to explain why and how to fix it.

Converting Remember back when I said C# is a type safe language? Well, when attempting to populate our `int 'width'`, we called `ReadLine()` but it returns a string of the user's input.

There are two main ways to work between types:Convert
Cast

Converting, as the name suggests, attempts to convert the input into the desired type. Casting assumes the data is already the correct type but it just needs to be realized as one. This is an important difference for down the road.

We can convert our input like so:

```
Code string input = Console.ReadLine();  
int width = Convert.ToInt32(input);
```

We're calling the `Convert` class (from `System` namespace) and specifying we want an `int`. Think back to the data table, an `int(eger)` is a signed 32-bit integer, hence '`Int32`'. Think of `int` as an alias to `Int32`.

It's also safer in the sense that it raises the opportunity for exceptions. Casting certain types does not. For instance, say we wish to cast an integer into a byte (unsigned 0-255). We could cast like so:

```
Code int src = 256;  
byte dest = (byte)src;
```

However, `src` is out of the byte storage range, our `dest` will hold 0 (range is offset by 1, more on that later). Converting (`Convert.ToByte(src);`) would throw, in this case, an overflow exception since the source was out of range.

We will now perform conversions for both width and height. You can reuse the input string to read input again for the height.

Collapse if you're having trouble ([Click to View](#))

```
Code Console.Write("Enter the width: ");
```

```
string input = Console.ReadLine();  
int width = Convert.ToInt32(input);
```

```
Console.Write("Enter the height: ");
```

```
input = Console.ReadLine();  
int height = Convert.ToInt32(input);
```

You'll notice in there I called '`Write`' instead of '`WriteLine`'. The difference is that the latter writes the output and terminates the line i.e. carriage return. The former just writes to the console, hence the space. Calling `ReadLine()` will read and terminate the line. It will look a lot cleaner, try it yourself!

I'm also writing a space at the end so the input isn't squashed next to the instruction.

The next part is to plug these values into the perimeter formula.

Naming Since variables can be aliased as we please, it's important that we know how to choose and name our variables in a way that the data they're holding can be easily identified. To keep it brief and so as not to insult your intelligence, give the list below a read.

Camel cased `int firstAge`; - first word lower, second identified by its title-case

Easily readable and identifiable. `int age`;

Don't shorten to the point where it hinders the above. `int a`;

Avoid association with core keywords/identifiers `int publicAge`;

Never use Hungarian notation `int intAge`; - prepending the contracted (if necessary) data type

Classes, namespaces and methods should be pascal-cased (`class ConsoleApplication`). For now, that's essentially it. The casing difference between naming makes such a huge impact on readability, you'll have a hard time reading someone's code which hasn't been written adhering to these guidelines.

Operators By definition, an operator is a character that represents an action (operation) on any number of subjects (known as operands). There are many in C#, but we will only look at the core numeric operators in this chapter. Others will be introduced when they come into play.

You'll find these are near identical to real world mathematics. Arithmetic operators include:

- Additive+
Adds one operand to another

- Subtracts one operand from another

- Multiplicative*Multiplies two operands together

- /Divides two operands

- %Known as modulus, it computes the remainder from a division between two operands

- Primaryi++Returns the existing value and then increments by 1

- i--Returns the existing value and then decrements by 1

- Unary++iIncrements by 1 before returning the (updated) value

- iDecrement by 1 before returning the (updated) value

- Assignment=Assigns the right operand to the left

- +=Increments by the right operand

- =Decrement by the right operand

- *=Multiplies by the right operand

- /=Divides by the right operand

- %=Performs a modulus on the target by the right operand, sets the value as the result

All but the first of the assignment operators can be considered as contractions of sums. For example:

Code `int i = 1;`

`i += 1;`

Produces exactly the same (`i = 2`) as:

Code `int i = 1;`

`i = i + 1;`

Which can be shortened by:

Code `int i = 1;`

`i++;`

Back to our rectangle project, the formula to calculate the perimeter (sum of all sides) is: $P = 2(L + w)$, L being length and W being width. This can be represented as $(L * 2) + (w * 2)$ and it's how we'll calculate it in our project.

You'll find that a lot of programming involves plugging values into formulas and it's no different to mathematics to be less alien.

Apply this formula by creating a new variable of type `int` and name it 'perim'. After writing in the formula, press Start to debug and we will try it out.

My test values of 5 and 3 should produce 16, and it does!

Your code should look something like this:

Code string input;

```
Console.Write("Enter the width: ");  
input = Console.ReadLine();  
int width = Convert.ToInt32(input);
```

```
Console.Write("Enter the height: ");  
input = Console.ReadLine();  
int height = Convert.ToInt32(input);
```

```
int perim = (width * 2) + (height * 2);  
Console.WriteLine("Perimeter: " + perim);
```

```
Console.ReadLine();
```

I've written input as an empty string at the top to demonstrate that primitive type variables don't always require a value on declaration. This changes as we move on to more complex objects, but more about that later.

Commentary An integral part of programming is internal commentary. Before I explain, let me show you an example of a comment:

Code //Calculate perimeter

```
int perim = (width * 2) + (height * 2);
```

You see "Calculate perimeter" prefixed with a double forward slash? That's called a comment. They have a variety of uses, including:

- Describing code

- Ignoring code

- Inline developer notes (TODO, BUG, etc.)

- Information for other programmers using your code

It doesn't matter what you write, from actual code to a Harry Potter book, the compiler will not

recognize it as valid instructions.

Although it's good practice to document your code with comments, don't go overboard with it.

Things such as:

Code //Read console into input

```
input = Console.ReadLine();
```

Where a block of code's purpose is clear, commentary isn't needed. This stuff is designed to be readable and comments should serve as a boost.

The double slash ("//") is used for single line comments, but what if we need to comment multiple lines? There is a solution for that, multi line comments look like this:

Code /*

- * A line

- * Another line

- * One more line should do it

*/

The top line ("/*") can also be used to comment, it comes down to preference. The asterisks are also optional. Multi line comments are useful for commenting out blocks of code, for example:

Code /*

```

Console.WriteLine("Enter the width: ");
input = Console.ReadLine();
int width = Convert.ToInt32(input);
*/

```

Will be recognized as comments. You can identify commentary by its solid green color.

Syntax Highlighting Speaking of colors in the code editor, it's time we took a look at syntax highlighting. Although this chapter could be grouped in the IDE chapter, explaining its uses and giving examples may have thrown you off. But you've lasted this far, so good job!

You'll have noticed that your code sports a few colors. I'm using the dark theme, so my code looks like this in the editor:

- Lets break down these colors.
- blue Reserved for core keywords of the language. Things such as primitive types, access modifiers, <1 char operators (typeof for example).
- pale green Identifies classes and objects.
- terracotta Exclusive for pre-defined (a.k.a hardcoded) strings
- dark green Marks commentary, both single and multi line
- pale green Numeric values, 1 char operators (+, -, etc.) and enum (you'll learn about these later)
- name highlighting

These colors will change with your theme or you can install custom coloring packages, but the laws of coloring above will always remain unless you disable syntax highlighting.

Odus - February 19, 2016, 12:23 pm

Beginner Boolean Logic There are three main elements to programming: sequence, selection and iteration. We've already looked at sequence (line-by-line), now we'll look at selection. But before we can go straight into it, we need to look at something called boolean logic. If you've done any intermediate circuitry work, this will be easy for you.

As the name suggests, selection involves making choices. So far, all our work has been one direct path from top to bottom. Selection gives us the opportunity to branch into a different code path depending on certain parameters. What these are, is our decision.

Back in the Fundamentals section, we looked at primitive types and one named 'bool' was there, representing true or false. We can compute what are known as expressions in order to give us this boolean logic. Declaring is just the same as before, it looks like this:

Code bool valid = false;
Using this kind of data requires a new set of operators. Like before, a chart is below showing what you'll need to start with.

Equality==Reference equality1 == 1 = true
!=Reference inequality1 != 1 = false

Relational<Less than1 < 2 = true
>Greater than1 > 2 = false
<=Less than or equal to1 <= 1 = true
>=Greater than or equal to1 >= 2 = false
ConditionalAND&&(true && true) = true
(true && false) = false
(false && false) = false

```
OR||(true || true) = true
(true || false) = true
(false || false) = false
XOR^(true ^ true) = false
(true ^ false) = true
(false ^ false) = false
```

XOR (exclusive-or) works a little different than the rest. It must evaluate both operands before it can draw its conclusion, the rest are known as short-circuit operators. If the first will cause the resultant to fail, it will short the block and not bother evaluating any subsequent conditions.

There's also one other operator which is known as negation. It negates the expression/bool and is simply an exclamation point like the reference inequality operator. For example:

```
Code bool valid = false;
valid = !valid;
```

In that sample, our bool is being given the opposite value of itself. It's currently false, so negating that will turn it to true.

In this chapter, we will only study the equality set, the rest will follow in the next chapter. Remember that the reference equality operator is different from the assignment operator (=). It makes distinguishing between them much easier than in other languages which don't support it. For example:

```
Code //assume 'pin' is of type string
bool validPIN = pin.Length == 4;
```

This expression computes the length of the string 'pin' (we mentioned the Length property briefly before) against 4, since PINs are generally 4 digits in length.

If we entered "1234", our bool named 'validPIN' would turn true. If we entered anything below or above 4, it would return false. We could be more diverse in our length checks, how about we wanted to validate a minimum length? Also simple enough with our relational operators. Like so:

```
Code bool validPass = password.Length > 7
```

This will require that the password length is at least 8 (greater than 7) characters in length. Equally, that same expression can also be represented as:

```
Code bool validPass = password.Length >= 8
```

In situations like this, it comes down to preference, but each have their own uses which you'll find out later.

Conditionals In the previous chapter, I mentioned that selection was a part of programming. Boolean logic was first, this is the second half and we start with what's known as an If statement. These can be used to make a selection based on logic that we studied above, and the program can branch depending on the result. We'll take a look at this by creating a program informing the user if they are or above the legal driving age.

The first part is up to you. Create a constant int with your country's driving age. For example, it's 16 for a learner's permit in the USA so your constant's value will be 16. You then want to ask the user if they are over this age, but don't read the input just yet.

If you're having trouble or want to check, the solution is here. [\(Click to View\)](#)

```
Code const int legal_age = 16;
```

```
Console.Write("Are you over " + legal_age + "? (Y/N) ");
```

Keep in mind the naming conventions we spoke about earlier. Your programs should be highly

maintainable, meaning it's quick and easy to update without breaking too much (if anything). This is why we've opted for a constant.

This is how we could read the input:

```
Code char answer = (char)Console.Read();
```

We're opting to read the answer as Y(es) or N(o). You've seen Write and WriteLine, the same applies for the Read set. Calling Read() on its own reads only the first character from the stream. Because of this, we can safely cast instead of having to call from the Convert class like we did earlier.

However, it leaves after reading the first character, leaving the rest of the line awaiting consumption, which ReadLine does, leaving no further instructions and terminates like we spoke about right at the beginning. An easy fix is to call 2x ReadLines, but we'll instead switch to ReadKey, which looks like this:

```
Code char answer = Console.ReadKey().KeyChar;
```

Calling ReadKey alone returns a reference type ConsoleKeyInfo which allows access to various attributes of the console key such as modifiers. We only want the actual Unicode character that was pressed, so we must read from its KeyChar property as you can see from above.

But we're not out of the woods yet. This call is made the moment a key is pressed and we want the user to submit it themselves by pressing Enter. Calling ReadKey doesn't terminate the line either, so to kill 2 birds with 1 stone, we'll call ReadLine just below it.

If you're not feeling confident, skip this paragraph and move straight to the next paragraph, else keep reading. If you run this, you'll see that the first key is now part of the output stream and cannot be undone. We can resolve this by going back to our old friend, ReadLine. Since it returns a string and strings are essentially an array of chars, we can read from index, meaning we can read each character of the string individually. After calling ReadLine, append "[0]" to the end, but before the semi-colon (line terminator). This will read the first position since collections start at 0 in programming, but more on that in the Arrays chapter.

Now it's time to evaluate this char using an If statement, which looks like this:

```
Code if (<condition>)
```

```
{
    <true instructions>
}
else
{
    <false instructions>
}
```

The first part, <condition> must be a logical expression, meaning that it returns a bool. If this condition evaluates to true, everything inside the true instructions braces will be executed, if not, everything in the false instructions braces will be executed. These braces follow the same principle as braces in namespaces, classes and methods.

The else block is optional.

To put this into practice, we'll use the reference equality operator (==) against the char:

```
Code if (answer == 'y')
{
    Console.WriteLine("You are old enough to drive.");
}
```

However, this will only fire if the input is 'y', reading it as 'Y' wouldn't satisfy the condition. We could make it lower case by default, but this is the perfect time to introduce conditional operators.

Conditional operators (AND [&&], OR [||], XOR [^]) allow us to combine multiple expressions into one final condition, which will be used to decide the path. Since we wish to compare both cases, we'll select the OR operator for this task. Like so:

```
Code if (answer == 'y' || answer == 'Y')
{
    Console.WriteLine("You are old enough to drive.");
}
```

Now we need to account for N submissions. Although introducing an else block would do the trick, it would also accept any other character as a contender for not N. This is fixable through use of elseif. Which looks like:

```
Code else if (answer == 'n' || answer == 'N')
{
    Console.WriteLine("You aren't old enough to drive yet.");
}
```

And is added directly below the if statement. Should the original if fail, flow will move to the else-if block and evaluate. So at this point, your code should look like this:

```
Code const int legal_age = 16;

Console.Write("Are you over " + legal_age + "? (Y/N)");
char answer = Console.ReadKey().KeyChar;

Console.ReadLine();

if (answer == 'y' || answer == 'Y')
{
    Console.WriteLine("You are old enough to drive.");
}
else if (answer == 'n' || answer == 'N')
{
    Console.WriteLine("You aren't old enough to drive yet.");
}
```

```
Console.ReadLine();
```

There is just one thing we haven't accounted for, and that's unaccounted-for chars. Anything other than Y/N is invalid so we should handle that as anything else. You guessed it, it's an else statement, which looks like this:

```
Code else
{
```

```
}
```

As you can see, there's no condition that must be met, code will only fall to this block if none of the previous conditions are met and this is what we're looking for. It works identically to if/elseif, so place a WriteLine call that informs them their input is invalid.

Switches In the previous chapter, we examined conditionals through the If statement. Now, we'll take a look at its 'brother', the Switch. Don't worry, it's nothing near as complicated as it may sound. This is how it looks:

Code switch (subject)

```
{
    case <value>:
        <action>
        break;
    case <another value>:
        <another action>
        break;
    default:
        <failed action>
        break;
}
```

Similar to If, it also represents a crossroads for logic as you can see above, but works a bit differently in its use. Unlike If, the Switch is incapable of computing cases as expressions, they are taken at face value. For example, you couldn't write:

Code switch (someInt)

```
{
    case someInt < 5:
//...
```

It must be directly comparable, think reference equality operator. The keyword 'break' is bolted to the bottom of each condition action block in order to break from the switch. Failing to include this means it will fall through to the next condition, which is how we can use multiple cases against one action block. It's also there for readability's sake. Remember, C' is modeled from the C family and Java.

The last part of it is 'default' which is identical to 'else' of the If set. Should no case fit, it will default to this block. It's not mandatory to actually do anything other than include a break keyword, but it's good practice and is enforced by some similar languages. Consistency!

If we converted our code to include a Switch rather than an If, it would look like this:

Code switch (answer)

```
{
    case 'Y':
    case 'y':
        Console.WriteLine("You are old enough to drive.");
        break;
    case 'N':
    case 'n':
        Console.WriteLine("You aren't old enough to drive yet.");
        break;
    default:
        Console.WriteLine("You haven't entered a valid answer.");
        break;
}
```

As you can see, stacked case conditions fall through to the action block, similar to the OR (||) operator in conditionals. You may also hear that these can be faster than If statements. The truth is, they can. But not always. It comes down to variable density and depth. It may not even compile to what a switch should compile to (jump table), but a tower of ifs. So don't immediately favor this against ifs. Also keep in mind lack of condition testing as cases.

Iteration Back in the Boolean Logic chapter, I mentioned there are three main elements to programming: sequence, selection and iteration. We've looked at sequence, covered selection above and it's time to look at iteration.

A loop is a statement, or set of statements, repeated a specified number of times or until some condition is met. The type of loop you use depends on the task at hand, these are the main two types:

For
While

The first operates using an iterator (index) variable which is increased on each lap of the loop. The second works using an expression which is computed each iteration and control is only transferred outside the loop when it evaluates to false. We use loops in order to avoid repeating ourselves over and over again in terms of code. It also means we can make small changes on each iteration.

Taking a look at the For loop, it's organized like this:

Code for (<iterator> = <initial value>; <condition>; <increment>)

```
{  
    <instructions>  
}<iterator>
```

A loop control variable that is declared for the loop and initialized once

<initial value>Starting value of the above variable

<condition>An expression which determines how long the loop will run, usually involved the iterator variable

<increment>Called after each iteration, usually to update the loop variable

An actual For loop would look like this:

Code for (int i = 0; i < 5; i++)

```
{  
    Console.WriteLine(i);  
}
```

In flow terms, this is the order in what's happening:

Loop variable is declared and initialized once

Condition is evaluated and will execute one iteration if it returns true

Instructions in the loop block are executed

Control update/increment takes place (i++, remember incrementing by 1?)

Steps 2-4 are repeated until the evaluation returns false. Running this, the console will print:

Quote:0

1
2
3
4

It's iterating 5 times, but when control variable i is incremented again and returns as greater than 5, the loop is exited. Our iterator has started from 0 and only gone up to 4. It's important you note that is still 5 iterations, this is crucial for the next chapter.

Moving on to the While loop, it looks like this:

Code while (<condition>)

```
{  
    <instructions>  
}
```

Unlike the For loop, this one keeps track of no iteration and it's possible to go on forever (what's known as an infinite loop) easier than its counterpart. It will run so long as the condition evaluates to true.

As an example, we'll convert our For loop into a While loop:

Code int i = 0;

```
while (i < 5)
{
    Console.WriteLine(i);

    i++;
}
```

It will print exactly the same thing (0-4 each on a new line), only we're incrementing ourselves. Removing 'i++;' would continuously print "0", an infinite loop since variable 'i' will never increment, so it will never exceed 5, therefore no opportunity to break.

A difference between these is that variable 'i' is accessible outside the While loop, we can continue to use it and do whatever we want with it. It's value will be 5 should we read from it after the loop. For, however, limits this variable only to what's inside the loop. This concept is known as "scope", meaning at what level something can be accessed at, and it applies to everything. If statements, loops, switches, methods, classes (sort of), etc.

We can demonstrate this loop using our driving age calculation, looping until valid input has been made. Create a variable of type bool and name it as 'validInput', initialize it as false. Then create a while loop just below asking the question and above reading the input. Test the negated value (! operator) of this validInput bool. Surround everything before the ReadLine stall in braces as it's in the scope of the loop.

Before the break keyword of Y and N cases, set validInput to true, so it knows to break from the loop. If you're struggling, collapse this for an example of my code. [\(Click to View\)](#)

We're not bothering to set validInput to false in our default-case instructions since it's already been set and there's no place it could have changed other than inside valid input, in which case the control will transfer outside the loop.

Take it for a spin and you'll notice it will continually prompt for an answer until it gets something valid. Awesome!

The While loop also has a sibling of its own, called the "Do-While". It looks like this:

```
Code do
{
    <instructions>
} while (<condition>)
```

The key difference is pre and post conditional checking. I've drawn a diagram below to explain the difference. Control flow for the original While goes like this:

Evaluate condition

Execute block if true, exit if false

Do While, however, goes like so:

Execute block

Evaluate condition

If true, go back to step 1 and execute, if false, exit block

With the first, condition is evaluated before the first iteration, so what's inside the loop block may never be executed. The second performs one full iteration before performing the check, so at least one lap is guaranteed.

Collections Now that we can repeat code with loops under our belt, I want to ask you this: What if

we wanted to determine ages for a group of people? Writing something like:

```
Code int firstAge = 18;  
int secondAge = 27;  
int thirdAge = 15;
```

...

Would get very tedious. It also means we're limiting ourselves to a set number. This is where collections come in, we'll be studying the array in this chapter.

Arrays are simply a group of types at a fixed size. We can refer to them by the same alias but supply an index to get a value at a position in the array. It sounds complicated, but let me show you...

Declaring an array looks like this:

```
Code int[] ages = new int[5];
```

Here we're declaring a new int by the name of 'ages', but you'll notice two square brackets bolted onto the end of the type. This is how we can identify an array and you've just covered the third and final brace! The first; regular parenthesis, is for methods and containing args, the second; curly braces, are markers for classes and methods and whatnot, square; what you've just seen now, is used for indexing on collections.

Where it's initialized, we see the 'new' keyword, which means we're spawning a new instance of an array of type int. In the square brackets is the number 5, so this array 'ages' has the capacity to store 5 individual values all under a single roof.

We can populate these positions like so:

```
Code ages[0] = 18;  
ages[1] = 21;
```

...

Now you'll notice I started at position 0. Back in the collections chapter, I mentioned that knowing from 0-4 was a total of 5 steps being crucial, and this is why. Arrays are 0-based, meaning they start from 0. The best analogy I can give is to think of a building at the ground floor. You're neutral at the first position. Going negative would mean you're going underground. At 1 would suggest you're 1 level above i.e. on the first floor.

So if we attempted to populate position 5:

```
Code ages[5] = 50;
```

It would crash, giving us an out of bounds exception since we're attempting to access a position larger than the array's capacity/size. [4] would be the last position, but it's still 5 steps from 0.

Now this is the fun part where we merge iteration with collections. That For loop supplies indexing and we'll use it to communicate data to our array. Remember how to read an int from the console?

```
Code int input = Convert.ToInt32(Console.ReadLine());
```

We're going to use that same line. Scrap your existing code except the final ReadLine method & ages int[] declaration. Create a For loop with a variable named 'i' as the iterator then set its value to 0. For the condition, i should be less than the ages array's length (think .Length property) and we want to increment i by one each iteration.

Solution ([Click to View](#))

```
Code for (int i = 0; i < ages.Length; i++)
```

```
{
```

```
}
```

We could equally use the condition as $i < 5$, but comparing to the array's length allows us to change the size of the array in the code without having to alter the array condition as well. This is why we're using the Length property of the array.

To prompt the user, we want to write "Enter age for position ", concatenated with i and followed by a colon (:) & a space for padding. Read the entire line as input to terminate the line for the next iteration.

Can you think on how to access a position in the array? Variable 'i' is an int and represents the current iteration, we use square brackets at the end of the alias to qualify the index.

Your code should look like this:

Good job if you arrived here yourself. At this point, the user's input has been read into an array. We're now going to iterate through once more and process it. Since we're reading ages instead of matching chars, we now have some mathematical work to do. The value at the position of `ages[i]` must be greater than or equal to than our driving age (`legal_age` constant).

We want to test this using a conditional. If it returns true, print:

Quote:User at position {current position} is old enough to drive.

Else, we print:

Quote:User at position {current position} is not old enough to drive yet.

To achieve this, think on string concatenation (hint: +) and how we access a position's value in an array. This is what you should have:

Now we're going to go one step further. If the user isn't old enough to drive yet, we'll calculate how far off they are from the legal driving age. The formula for this is to subtract the `legal_age` constant from the user's age. Store this value in a variable of type int named 'difference'. Adjust your console output to this:

Quote:Console.WriteLine("User at position {current position} is not old enough to drive yet. They will be old enough in {difference} years.");

Running this against ages: 10, 21, 17, 16, 12 should output this:

Quote:Enter age for position 0: 10

Enter age for position 1: 21

Enter age for position 2: 17

Enter age for position 3: 16

Enter age for position 4: 12

User at position 0 is not old enough to drive yet. They will be old enough in 6 years.

User at position 1 is old enough to drive.

User at position 2 is old enough to drive.

User at position 3 is old enough to drive.

User at position 4 is not old enough to drive yet. They will be old enough in 4 years.

If not, this is how it should look:

There's just one loop left I've left until last and that's so you'd understand it better after some practice. It's called `ForEach` and is a version of the `For` loop. It looks like this:

Code foreach (<type> <variable> in <collection>)

```
{  
    <instructions>  
}
```

You can see the similarities, but this is what's different:

type	Type of the iterator (int, string, etc.)
variable	Loop variable
collection	An array of some sort to iterate through

In this, we don't reference by index. We reference using our iterator variable which progresses to the next value in the collection with each iteration. If we don't need to use indexing anywhere in our loop instructions, we can use this for a cleaner and easier to read loop.

If we switched out our current For loops to ForEach, it would look like this:

You'll notice that only the second loop was changed. This is because the iterator variable is read only. In most cases, the iterator type will match the type of the collection, though this isn't enforced. We can't update the position's value from it. Since there's no position, I also reworded the console output.

Methods Hopefully you've had some practice at the fundamentals and beginner level concepts. It's now time to move on to more intermediate stuff and we're going to dive right in with methods.

Until now, we've only practiced inside Main, the entry point method of our application and that they must be placed in some sort of class. Now we're going to try out working with other methods. There are two types of methods:

Static

Instance

The first is identified by the 'static' keyword which is also seen in Main, the latter isn't. We've already called plenty of static methods such as Console.Read/Write(Line). But there are methods which can only be accessed through an instantiated (instance) object. For example, string has a method named "ToUpperCase" (converts all letters of the string to their upper case format) which can only be accessed on a string.

We will look at something called a sub routine, identifiable by its keyword "void", meaning that it doesn't return anything, which we covered earlier. This is ideal for calling a set of instructions by one name. Similar to a variable, we can call it over and over again but only having to write it out once. It can simplify your code big time. For example, let's look at the Console.WriteLine sub routine.

```
Code public static void WriteLine(string value)
{
```

```
}
```

Analyzing each keyword: public At what level it can be accessed at, more of that in the next chapter.

static Static keyword means it's a static method

void Return type. Void returns nothing, but we can also switch this with string, int, bool, etc. to return a value of that type

WriteLine Name of the method. Its function should be clear and concise

string value Parameter type and alias, it's a value we pass to the method that will be used. In WriteLine, this value is written to the console

Sticking with our previous code, which is this:

```
Code const int legal_age = 16;
int[] ages = new int[5];
```

```
for (int i = 0; i < ages.Length; i++)
{
```

```
    Console.Write("Enter age for position " + i + ": ");
```

```

    ages[i] = Convert.ToInt32(Console.ReadLine());
}

foreach (int age in ages)
{
    if (age >= legal_age)
    {
        Console.WriteLine("User aged" + age + " is old enough to drive.");
    }
    else
    {
        int difference = legal_age - age;

        Console.WriteLine("User aged" + age + " is old enough to drive yet. They will be old enough in
" + age + " years.");
    }
}

```

Console.ReadLine();

We'll create a new static method with a return type of void and name it "PromptForAge", taking no parameters (empty parenthesis).

```

Code public static void PromptForAge()
{

}

```

This is written just below the closing brace of the Main method, it's still inside our Program class, so the access modifier (public) can be safely changed to private, more on that in the next chapter.

In this method, Write (but don't terminate the line) to the console:

Quote:Console.Write("Enter age: ");

Now take out the first line where you're writing this to the console inside the For loop and begin to write "PromptForAge". You'll see IntelliSense, the little window that guesses what you want to write, start to bring up your method. Press enter or tab to complete. You must fill in the parenthesis and terminate the line (semi-colon) yourself.

Run and see that this method is called, printing to the console. Awesome! The only difference is that we can call this method outside the loop, even if we don't need it, it's there anyway.

To give you a more practical example, we'll rewrite our legal age determination into a method with a return type of bool, this kind of method is known as a function and is written exactly the same, only 'void' is switched with the return type. Name it "CanDrive" and accept a parameter of type int and name it "age". Look back at the Main method for passing args.

It should look like this:

You'll notice the squiggly red line highlighting an error. It reads:

Quote:Program.CanDrive(int)': not all code paths return a value

It's right. We're not returning anything, but don't worry, we'll fix it now. Since we're moving the age calculation into a method, move the legal_age const in with it as we won't need it inside the Main method. Now we want to create a bool which is valued on the age parameter being greater than or equal to the legal_age const, just like in our Main method.

It should look like this:

It's possible to omit the storage bool and return directly:

Code return age >= legal_age;

But for demonstration, I chose to go the long way. Now, replace the age difference calculation inside the If condition with a call to CanDrive, supplying the age iterator variable from the loop as an argument. Like so:

Now there's one error left:

Quote:The name 'legal_age' does not exist in the current context

This error is thrown because we have moved it to a method. But we're also going to move the age calculation into its own method too. Create a method named "AgeDifference", accepting an arg of type int named 'age'. We will return the difference between the legal_age const and the parameter sent. Copy the legal_age const into this method as well.

Now let's head back to our Main method and remove the variable 'difference'. Replace where we reference this variable in the WriteLine method with a call to AgeDifference, supplying the loop variable 'age' as an argument:

Code AgeDifference(age)

As you can see, methods can be called anywhere, even inside an operation such as string concatenation. Your else statement should look like this:

Since we now have the age calculator in a method, we could even show how many years ago a person over the legal age was able to drive. But this would return a negative number, so you would have to negate it back to positive before displaying. Attempt it if you wish.

Odus - February 19, 2016, 12:23 pm

Intermediate Globals After implementing our two (well, three) methods, you might notice that we have 2x of the same constant 'legal_age'. We've kind of defeated the purpose of constants in the fact that they represent one value and is mapped throughout the program, so once change would affect everything.

I'm pleased to say there's a solution of this. Thinking back to scope, a variable is only accessible in the same scope it was declared in. So what if we moved it up a level into the class? It's absolutely possible. Shall we do it? Yes!

Take out the consts from the AgeDifference and CanDrive methods. Between the opening brace for the Program class and just above the Main method, paste it in. You'll see that all errors have suddenly disappeared. We've just declared the legal_age const in the class, so everything in the class scope (including methods and their sub-scopes) can access it. But we're not out of the woods yet, there are still some conventions & practices to adhere to.

Rewrite your const with a private modifier, just like a method so other classes can't access it. We also want to rename it like so:

Quote:LEGAL_AGE

It's fully capped as a naming convention that global consts are fully capitalized and spaced with an underline. I'm guilty of this convention, others will have different preferences but this form is used in many languages. Since C# is case specific, renaming manually would mean we have to rename each reference to it in methods and whatnot.

Fortunately, Visual Studio, the glorious IDE that it is, has provided us a shortcut. Hold Ctrl and double tap the R key, the alias will highlight green and you can retype as "LEGAL_AGE". All references to it throughout the entire program will be updated.

However, global constants aren't always a good thing. Remember that everything in the class (or even outside) now has access to it and as a programmer, you want to keep things scoped as closely as possible. This could even include passing it as a parameter.

If we were using a global variable (just like local, but same rules as global constant), these methods are free to change it as well. So avoid them when you can, this was just to show you they exist and how to use them.

Parameters Supplementary to methods are what's called parameters, also known as arguments (args). A method with parameters will take arguments. That's pretty much the difference.

Think of parameters as variables with values already assigned to them that we use in our program. Cast your mind back to when we looked at calculating the perimeter of a rectangle. To be able to access this same $2(l+w)$ formula from other methods, we could use 2x globals, but this is excessively scoped. Meaning that other methods that won't potentially need access to the length and width of the rectangle now have access to it.

Remember when we said a lot of programming involves working with data we don't know? This is one of the best examples I can give you. So we don't include globals, create a function that returns an int, and accepts 2 arguments (note the wording difference) for length and width, both of type int. We'll end up with something like this:

```
Code private static int CalculatePerimeter(int width, int length)
{
    return (length * 2) + (width * 2);
}
```

I'm returning the product of the expression on the same line, which is just the same as writing:

```
Code int result = (length * 2) + (width * 2);
return result;
```

We are able to use these parameters just like local variables as well, they're not constants.

This is known as passing by value, where we pass the value of the object to the method and the variable passed will not change. The other is passing as a reference (NOT the same concept as reference types) where it can be used and manipulated by the method.

To demonstrate, we'll write a method named 'MultiplyWrite', accepting a parameter of type int but as a reference and named 'operand' - since it's the operand of this operation. It's identified from the 'ref' modifier and looks like this after multiplying and printing:

```
Code private static void MultiplyWrite(ref int operand)
{
    operand *= 3;
    Console.WriteLine(operand);
}
```

Run it and it will input 15. Print out the starting variable named 'number' after calling MultiplyWrite and you'll notice it was also changed:

Cool, right? But you'll almost never need to use something like this, only when working with multiple returns which we'll dig into in the Advanced section.

Classes If you've enjoyed yourself so far, you're in for a real treat! We're about to get started on the Object Oriented Programming (OOP) paradigm. This is one of the most important aspects of intermediate programming, so we'll go over what it actually is before we use it.

OOP is based on the concept of 'objects', which contain data (known as attributes) and procedures (known as methods), together known as members [of an object]. Using the Console class which we're familiar with, it treats the console as an object.

We can write to it and read from it using Write/ReadLine, which are methods of the Console class. It also contains attributes which provides information about itself, as well as giving us the option to change it. For example, we can get or set the background color of the console by using its BackgroundColor property. We can also change the title of the console like so:

```
Code Console.Title = "My Console";
```

Just like methods, we go down the chain using the period (.) which gives us access to an object's members. Objects don't have to always represent actions, they can be used simply as a data structure used to profile data on a subject.

I think I'm starting to sound a bit textbook with this, so let me demonstrate on a data model for a student. Right click on your project in the solution explorer (CSConsole) -> Add -> Class. Name it 'Student' and create it. You'll see it looks very similar to the program's main class.

We start by writing the skeleton of our class:

```
Code public class Student
{
```

```
}
```

You've seen these keywords before, the only difference is the name of the class which is Student. Now it's time to think on what kind of information we would hold on a student. Name? Age? Test scores? Sure, we've used arrays before, let's write that in.

```
Code public class Student
{
    public string Name;
    public int Age;

    public int[] Scores;
}
```

Notice how there is no appearance of the keyword 'static' which we're used to now. This is because an object must be declared as one, if that makes sense. When we declare this, it is individual (like a real student). That's not to say we can't create many though.

Just because we're holding data doesn't mean we can't work with it internally. Classes support methods which can use its own attributes. For example, let's write a method to calculate the average score. We calculate averages by adding all elements and dividing by the answer by the number of elements. Since there's no hard limit on scores, we will use this data as it's held.

Inside a function named 'GetAverageScore', Create a variable of type int and name it 'averageScore'. Then, for each score in our public int array 'Scores', add it to the final score. Then, divide final score by the length of the Scores array. You should get this:

Remember the assignment operators? We're using them above. You'll also see the 'this' keyword which is used to refer to the current instance of the object. Notice it's semi-transparent/slightly darker. This is because it's optional (which it is), but to show you where these attributes are coming from, I've included it.

Now it's time to take our new class for a spin. Just like a type, we declare it like so:

```
Code Student std = new Student();
```

Then we can start to fill in the properties by referencing our variable 'std' of type Student. Just like in the other classes, we go down the branch using dot notation:

Properties The public variables in our student class are exactly that: variables. We're now going to look at properties in a class. The difference at face value looks like this:

The keywords here (literally) are get and set, known as accessors. Using properties gives us far more control as to how these properties can be used. We can moderate access, perform validation when updating values (null, less than X, etc.) and do underlying processing when setting values.

Good OOP practice will see you using a private variable to read/write to and the property acting as a relay. Like so:

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

With 'value' being the value passed when setting. As you can see, it's heavy on the lines if you keep a consistent syntax. Fortunately, C# has introduced auto-properties which means at compile time, these will be generated and implemented for you. If you're doing any kind of manual processing or want to relay something other than its value, you must take the manual route. So we'll use:

```
Code public string Name { get; set; }
```

Apply the getters and setters for each of your properties: Name, Age and Scores, then try it out. Results should be exactly the same... for now!

We're now going to adjust the scope of our Age property. Since it isn't humanly possible to change your age or date of birth. To prevent any accidental changes, we'll make this read-only. It's actually very simple, all we're doing is removing the set accessor, therefore removing possibility of change.

Like so:

```
Code public int Age { get; }
```

However, this introduces an error. Our existing code back in the main class is assigning a value to it. We still need to be able to set a value, how can we go about it? Find out how in the next chapter.

Constructors We're going to take a look at constructors to rectify this little assignment error going on in our project. Or more specifically: instance constructors.

A constructor is a method called when a class of some sort is instantiated. C# does not enforce a constructor, but the compiler includes one if no constructor exists in your class, and sets all values as default. First thing to do is remove the age assignment from the main class and get rid of that error.

Specific to our Student class, our constructor looks like this:

```
Code public Student()  
{
```

```
}
```

Empty as a drum. We're now going to use the constructor to get one-time access to certain values in our class, the only one in this case being the Age property. Thinking back to parameters, they're exactly the same in constructors. Also remember the use of 'this' keyword to refer to the instance.

This is how we'll set our Age:

As you can see, it's error-free. If we tried to set the Age property in our Student class anywhere outside the constructor, we'd get the same error as if trying to set it outside the class.

Going back to our main class, we update the declaration to provide an int as the age. Like so:

```
Code Student std = new Student(18);
```

It's important to keep the work in your constructor light, they should only be used to construct your class. Any heavy construction should be moved to an independent builder/factory method.

Enumerations Enums - short for enumerations - is essentially a grouped range of constants. It usually goes hand in hand with a type of some sort. It is declared like this:

```
Code public Enum Grades  
{
```

```
}
```

Then, any sort of name you like can be included, so long as they are delimited with a comma. Below the first brace of the CSConsole namespace but above the Student class, declare an enum which looks like this:

In reality, these are just int typed constants, so these values can be overridden should we need to use their values directly. They support int, byte, long and short (all signed/unsigned). You'll also notice the pale/minty green of the Grade. That's syntax highlighting again.

We're now going to implement a method inside our student class which will determine whether the student has passed or failed based on their scores. If the average score is above 70%, they have

passed (we're a generous university, after all) and if anything lower, they've failed.

It should return type Grade:

Since we're calling GetAverageScores internally and no longer need to know the average score outside of the class, we can actually make the method private, by changing the access modifier 'public' to 'private'.

Exceptions Occasionally, things may go wrong and we need to know how to deal with this. What you know as a "crash" is known to programmers as an "unhandled exception." Meaning that something unexpected happened we haven't managed to catch. Before we look, it's important to know that making your program 100% robust is ridiculously hard, and it gets progressively harder as your program increases in size & functionality.

To trap exceptions, we wrap potentially hazardous code in TryCatch blocks, which look like this:

Code try

```
{  
  
}  
catch  
{  
  
}
```

In the event that something inside the Try block throw an exception, it will divert to the Catch block, which gives you the opportunity to handle and possibly even recover from errors. We're going to demonstrate the latter in our student class.

So you get an idea of what a crash looks like when a debugger is attached, modify your code in the main class and take out the Scores setter, leaving you with this:

```
Code Student std = new Student(18);  
std.Name = "Odus";
```

```
Console.WriteLine(std.GetGrade());
```

Run it and you'll notice a pause in the WriteLine before it crashed and brings up the crash report. The window looks like this: [\(Click to View\)](#)

The exception was thrown inside the Student class, which is better for us since it gives us a chance to do the error handling backend. The TryCatch block allows us to catch specific exceptions, offering multiple paths of handling depending on the exception. First, wrap the method in inside the Try braces and place a Catch below.

We're also able to supply a 'parameter' for the exception thrown which tells us the type and gives us other information such as the description, stack trace and whatnot. But we won't in this case and you'll see why soon. Inside the catch block for a NullReferenceException, return -1.

Returning -1 means that all code paths in the method return some kind of value, and negative 1 is perfect since the minimum score is 0, and an average of 3x 0's will still give you 0, it's not possible to achieve a negative score (theoretically).

We need to continue handling this inside our GetGrade method. So we'll first add another const to our Grade enum named 'None'.

Code public enum Grade

```
{  
    Pass,  
    Fail,  
    None  
}
```

Back to GetGrade, we're going to catch exactly negative 1. In the odd chance a school using this would use a weighted grading system that can produce negatives, it will still handle it and the odds of -1 are extremely rare.

Since we're going to be evaluating GetAverageScore() more than once, we'll move it to a variable named 'averageScore' of type int and change our conditional to evaluate that instead of a call to the method itself. Between the first and else condition, insert an else-if clause that checked 'averageScore' for -1. If it does, return Grade.None.

This way, anything over 70 is a pass, anything -1 is no grade, and anything else is a fail. Your new GetGrade method should look like this:

As a final note, it's better to perform checks like this manually rather than relying on a Catch block to recover from an error. For example, checking if the Scores array was null would mean we didn't have to include a TryCatch block in our method. I used it here because it was a great example of accounting for errors.

Ternaries We're going to take a look at ternaries, which is essentially a condensed if-statement. Going back to our beginner project of being able to drive, don't worry I don't expect you to have retained that code, we done something like this:

Code if (age >= legal_age)

```
{  
    Console.WriteLine("You are old enough to drive");  
}  
else  
{  
    Console.WriteLine("You are old enough to drive");  
}
```

The difference is small, so we can take a shortcut by using a ternary, turning it into this:

As you can see, we get a fairly longer line, but it means we're not writing out a huge If statement which makes little difference in its result. Lets dissect the ternary:

Code (condition ? trueExp : falseExp)conditionIdentical to a logical expression in the standard If
trueExpExpression to execute if true
falseExpExpression to execute if false

As you can see, not very different, but saves a fair amount of space and in cases it's designed for,

means you write less. Although this is available to you, don't go overboard with it and especially don't ever nest ternaries. They work, but ternaries impact readability and even more so when nested.

This concludes our Intermediate section. The next will cover more advanced topics and it's only worth a read after a fair amount of practice on all previous sections. Good luck!

Odus - February 19, 2016, 12:23 pm

Advanced Parameters II We'll ease into more advanced topics by using the previous Parameters chapter as a bridge between intermediate and advanced. The same applies to the following two or so chapters.

Third type of parameter is called an out-parameter.

It works similarly to ref which we looked at previously, only that out is solely for projection, ref works for intake as well. There's no better example than TryParse, we'll use the Int32 (alias: int) one which looks like this:

Notice the difference in keywords, 'out' tells us that the variable's current value is irrelevant and plays no part in the method other than an assignment. If it were ref, the user (programmer) could be confused into thinking it mattered what they sent along with the call.

All being well, our result int would hold 123 and our bool would return true. This ties into exception handling as well, since avoiding exceptions is the even better choice as I covered earlier.

String Interpolation Until now, we've been using the concatenation operator (+) for our string operations. But what if we want to build more complex strings? A lot of concatenations will produce very long lines of code, even when split across multiple lines. Fortunately, we have a little thing called string interpolation.

C# 6.0 revisited string interpolation, building strings looks like this:

Notice the dollar sign prefix? This tells the editor that we will be building a dynamic string and to open up instance members when we insert curly braces. When executed, this will be written to the console:

```
Quote:Student name: Odus  
Student age: 18
```

I can tell what you're thinking: "How did it take a new line?" and it's simple. Look between them and you can see "\n". This is the next part of building strings and known as escape sequences. As you may have guessed, 'n' stands for new line. There are a huge bunch of these, but here's the chart:

<https://msdn.microsoft.com/en-us/library/h21280bw.aspx>

But what if we don't want it to recognize escape sequences? We can mark a string as literal instead of verbatim by prefixing the @ symbol to the string before the speech marks:

```
Code Console.WriteLine(@"test\tone");  
//Produces "test\tone" instead of interpreting "\t" as a tab
```

Inheritance Going back to the OOP series of chapters from the previous section, we're taking a look a concept called inheritance. Inheritance allows new classes to inherit the members of a

previous class. So when you subclass by making a new class that inherits from another, all its methods and properties become available to use in that class. Think of it as an extension. To demonstrate, we'll return to our Student class and add a constructor overload. The code for the Student class is below.

Code using System;

```
namespace CSConsole
{
    public enum Grade
    {
        Pass,
        Fail,
        None
    }

    public class Student
    {
        public string Name { get; set; }

        public int Age { get; set; }

        public int[] Scores { get; set; }

        public Grade GetGrade()
        {
            int averageScore = GetAverageScore();

            if (averageScore > 70)
            {
                return Grade.Pass;
            }
            else if (averageScore == -1)
            {
                return Grade.None;
            }
            else
            {
                return Grade.Fail;
            }
        }

        private int GetAverageScore()
        {
            int averageScore = 0;

            try
            {
                foreach (int score in this.Scores)
                {
                    averageScore += score;
                }
            }
        }
    }
}
```

```

        averageScore /= this.Scores.Length;

        return averageScore;
    }
    catch (NullReferenceException)
    {
        return -1;
    }
}

public override string ToString()
{
    return $"{Name}, aged {Age}, has an average score of: {GetAverageScore()}%";
}
}
}

```

Lets say this represents the attributes of a University student and this University wants to accept exchange students, but must record a log of the institution they've exchanged with and their own student at that institution. We could record this in a misc. notes field of the Student class, or we could inherit and subclass. To do this, create a new class file in your project (right click project -> Add -> Class) and name it ExchangeStudent. Keeping everything, you want to append the inherit marker and the class to inherit after the class name, like so:

```

Code public class ExchangeStudent: Student
{

}

```

We're now going to include properties for the exchanged institution and the student that was exchanged. That's right, we can even create properties of custom types/structures. Like so:

```

Code public string Institution { get; set; }
public Student Exchangee { get; set; }

```

If we return to our Main method and create a new ExchangeStudent and begin to populate it, you'll see that inherited superclass members (Name, Age, Scores, GetAverageScore(), etc.) are available as well as our newly set properties in the ExchangeStudent subclass.

Overloads Before we go any further, it's time we learned about overloads. An overload is a method sporting the same alias as another but accepting different parameters, and so doing (usually) slightly different work.

Using our perimeter function again as an example, this time it's overloaded to support types int and double:

Using it is no different than a regular method. Which method you are calling is even identifiable in the code editor. For example:

```

Code CalculatePerimeter(2, 2);

```

Has detected by the arguments that it will be calling the function that returns an int. If we were to introduce real numbers (irrational), it would infer its double counterpart:

```

Code CalculatePerimeter(2.3, 2.5);

```

One of the common places you'll see overloaded methods are constructors. Having multiple constructors available means the class can be more flexible in how it's used.

We're going to include another constructor in our Student class that takes a parameter of type DateTime named 'dob'. To overload, begin writing out the constructor again and fill in the code below. Your sibling constructors should look like this (subtracting 1 to assume no birthday yet):

```
Code public Student(int age)
{
    this.Age = age;
}

public Student(DateTime dob)
{
    this.Age = DateTime.Now.Year - dob.Year - 1;
}
```

When rewritten to accept our new constructor, it looks like this:

Overrides Continuing with the Student class, we're now going to talk about overriding methods which are different to overloads - don't confuse them! This will come more into play with more complicate OOP designs. Methods can be overridden to implement it yourself, whilst still retaining the method name that wouldn't be available since it's part of its inheritor.

For example, Object is the base class of pretty much all types, including our Student class. Object contains a method named ToString, which converts (or displays) the string representation of that object.

If we were to do something like:

```
Code Console.WriteLine(std.ToString());
```

It would only print its location in the namespace and name of the class. We must implement it ourselves in order to change it. If we wrote it as normal, a warning would flag informing us of this inheritance conflict:

When allowing IntelliSense to fill in the method for us, we get the default implementation:

```
Code public override string ToString()
{
    return base.ToString();
}
```

Now we can return anything we like when ToString is called. We'll use ToString as a default information print that returns:

Quote:"<Name>, aged <Age>, has an average score of: <averageScore>%"

Convert this to an interpolated string. Collapse the spoiler if you're having trouble. ([Click to View](#))

Quote:return \$"<Name>, aged <Age>, has an average score of: <averageScore>%"

Which if we run, would print:

Since the GetAverageScore member is private, we can still access it internally. Something like this may be good for debugging so we've included the student's average score.

File Input/Output You'll often hear the contraction "IO" which stands for input/output and this can be used to represent many things, but we'll be looking at local files. But before we can actually work with files, it's important to talk about what they actually are.

Until now, you've probably assumed files are unique in how they are written to storage. Files are actually all the same in the sense that they are just stored data. How it's formatted and its content is what makes it unique. Even something like a Word document file is a series of grouped files containing info about the document, such as its text, fonts, images, etc.

The important thing is that you realize files are unique in the sense that they're different. Weird way to put it, I know. So we'll only cover basic TXT files: files with no formatting or preservation, solely text.

When a file is read or written to, it's done via a stream. This stream represents a sequence of bytes being read or written.

We can harness the power of the .NET framework and use its classes for IO, conveniently located inside the System.IO namespace, so reference it at the top of your project:

Code using System.IO;

There are easy methods such as WriteAllText/Bytes/Lines, but we're going to be using the StreamWriter to write our files, giving us more control and since these methods use StreamWriter internally, it's better for us to see what's going on for a first look.

To get started, paste the code below and I'll explain what's going on.

```
Code string directory = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
```

```
using (StreamWriter sw = new StreamWriter(directory + @"\file.txt"))
{
    sw.WriteLine("test");
}
```

At the very top we've declared a string to fetch the desktop directory. These are part of the System.Environment class which gives us all sorts of tools for working with the local machine. SpecialFolder is an enum, we've worked with those before. Calling GetFolderPath and supplying it translates what the enum represents into a working, absolute directory.

Next we're opening a Using construct. These are used to mark the object for disposal after we've used it so as not to waste resources. Everything inside the braces is scoped to that StreamReader we've instantiated as 'sw'. It means closing the file is handled for us as well.

Inside the constructor for StreamReader we've passed a path. One of the constructors accepts this (more on the next chapter) and it will create the file if it does not already exist. Notice the literal (@) marker? We could do it without, but we'd have to use "\\" instead of a single slash.

Inside, it's easy to think of it as an external console. The same method [names] are available to us: Write, WriteLine, etc.

Working with it is very simple:

```
Code sw.WriteLine("Hello Desktop!");
```

Reading from a file a similar story, but there are key differences, particularly when it comes to processing large quantities of data. Instead of reading it all into memory at once using ReadToEnd(), we can loop through the stream until the end (as marked by The StreamReader's

EndOfStream property) and read each line individually. Like so:

Try it for yourself. It will read the same line "test" from the file. Those are the essentials of working with files. How you process them and what you do with them will vary.

Collections II XXX

Pointers Moving on to more advanced stuff, we're going to talk about pointers and unsafe code. When code is explicitly marked as unsafe, it is unmanaged, meaning it's compiled directly to machine code. Managed code (what we've been using so far) where lots of background management is present which looks after memory, ensures type safety, etc. All of this is considerably reduced if not omitted with unsafe code.

Using unsafe code is not recommended, but available should you be working very low level, complex PInvoke (which we'll learn about next) calls or when performance down to the nanosecond is crucial. None of which are really common.

Pointers are only available in primitive types, including: byte, short, int, long, char, float, double, decimal and bool. We lose string since it's not entirely primitive which we talked about at the start.

Enough lecturing, time to take a look. We must first set the unsafe flag for the compiler as it prohibits unsafe code by default. To do this, double click on Properties in the solution explorer, head to Build and check the "Allow unsafe code" checkbox:

When working with inner unsafe code, we must mark it within an Unsafe zone. Just like a Using construct.

```
Code unsafe
{
```

```
}
```

The unsafe keyword can be applied in local (using the above block), method and class level context. Marking an entire class as unsafe removes the need to qualify any unsafe methods or blocks inside safe methods.

Pointers essentially store the location of a value and serve to point to that address. Since we've yet to use the byte data type, we're going to test this out by halving a byte and creating a method to do so. Remember, since we're pointing to the value, we can update it directly, therefore we write this as a void.

You will notice the asterisk bolted to the end of the type, this is how we identify pointers. We continue to qualify with the asterisk inside the method, this time preceding the alias. Without the cast back to byte, that expression would produce an int.

Now to use this function. we'll set up a variable named 'b' of type byte, and give it a value of 6. Then, create an unsafe context inside your Main method and call HalveByte inside its body like so:

```
Code unsafe
```

```
{  
    HalveByte(&b);  
}
```

There is an ampersand before the alias, and that means we're representing the address of the byte, not its actual value. Again, this is only available in unsafe context. Your code should look like this:

Run it and you'll find that its value is 3. Now you're probably asking, why would we go to so much trouble? The difference, as I mentioned above, is performance. Here we are in almost total control with little intervention from any .NET runtime assistants.

This shows how pointers are being used here to update the value at its memory location (remember, that method is a void). However, it doesn't fully explain why it's marked as 'unsafe'. We're now going to take a look at printing an array in unsafe context.

Create an unsafe void named 'PrintArray', accepting a pointer of type int (not an array) and another named 'len' to represent the array's length. We're losing all .NET type safety with unsafe code, so there's no way to tell when to stop. Therefore we must supply the array's length from safe context.

We can then create a standard For loop, using an iterator variable of type int, named 'i', and running while it's less than our len parameter. This is what you should have:

We'll half it and print to the console like so:

```
Code Console.WriteLine(*(arr + i) / 2);
```

Parameter 'arr' representing the base address, and iterator 'i' as offset. Just like indexing (which is also supported, but again, we're going unsafe so I want to show you everything).

Attempting to use this would not work, primarily because of the error but also because we're using unsafe. We need to protect this array by reserving its location in memory.

To work with this, it's time to introduce another keyword: fixed. Since we're working with address only and not its actual contents, using this keyword stops memory management from transferring its location across the heap. When implemented, your final code will look like this:

Running this will print to the console:

Quote:1

2

3

4

All well and good. But this still hasn't shown us why unsafe is truly unsafe. Replace 'len' in the loop with the int: 10. It's well outside the bounds of our byte array (4 places). If you run it now, it will continue to call addresses on that offset. For example, I get this:

Nothing after 4 is inside my array. Now you can see that without type safety, it hasn't thrown an out of bounds exception and we're accessing memory locations that aren't entirely our own.

PInvoke Short for "Platform Invocation", this functionality allows us to call unmanaged external functions from a library (.dll) from our own managed application. It was developed back when the .NET framework was still new so it offered the same flexibility. As it matures, more and more external functions are being implemented to the framework so the likelihood of needing this lessens. But it's not fully covered, so we may still want to use it sometimes.

There's no need to set any markers for unsafe code this time since it's working externally, all we need is to reference System.Runtime.InteropServices.

In this example, we're going to sound a beep like when a message box appears on form applications. To do this, we must first reference where the function is housed. In this case, it's the user32 lib and looks like this:

```
Code [DllImport("user32.dll")]
```

```
static extern bool MessageBeep(uint uType);
```

This function is accepting a parameter for the beep type (alert, ok, exclamation, standard, etc.) but how did we know? There's a whole wiki dedicated to PInvoke and it is: <http://pinvoke.net/>

This site holds every discovered (and workin) PInvoker available for .NET and is user-contributed. Since documentation can be low or otherwise unavailable, it can be difficult to get it exactly right. What's available there is tried and tested. For example, our MessageBeep function is recorded here: <http://www.pinvoke.net/default.aspx/user32.messagebeep>

We've already discussed static modifiers, the new one here is 'extern', which indicates to the compiler that the method is external to the application. As you can see, this unmanaged function also returns a bool, likely representing its success. Its PInvoke article includes it in a class which contains an enum for the type of beep, but we'll take the standard one and supply 0xFF as an argument.

When written, your code would look like this:

As you can see, it's called just like a method from our own code. The applications of PInvoke are endless, spend some time on the wiki and you'll see just what it can offer.

Threading (Under Construction)

As of now, our applications have ran solely on a single thread, known as the main thread. If we're doing any intensive work, the thread will become occupied and since it also handles the UI, the UI will hang, giving the impression that the program has crashed. This is more visible in windows forms rather than console. This is where threading comes in, since we now have the opportunity to divide and conquer within our program (meaning break it into smaller parts!).

For this, we'll need to reference the threading namespace:

Code using System.Threading;

Going back to our trusty old-enough-to-drive scenario, create a method named CheckAge, with a parameter of type object and call it 'data'. This is the data (sometimes known as 'state') that's passed to the thread. It's mandatory to pass as object, but we can cast it back to int in the method.

I've wrote mine like so:

```
Code private static void CheckAge(object data)
{
    int age = (int)data;
```

```
Console.WriteLine($"You are no{(age < 16 ? 't' : 'w')} old enough to drive.");  
}
```

Notice the inline ternary? It's going to either print "You are now old enough to drive" or "You are not old enough to drive". Saves a lot of time typing out a full conditional for very little difference in results. Now to declare our thread:

```
Code Thread ageCheck = new Thread(CheckAge);  
ageCheck.IsBackground = true;
```

In this example, the thread's `IsBackground` property has been set to true, meaning it's a background thread. This kind of thread will terminate when its creator/the main thread terminates. Unlike foreground threads, which could still be running after the application has exited and can sometimes prevent it from closing.

To start, we can simply call `Start()`, which supports an overload that accepts a parameter to pass to our `ThreadStart`:

```
Code ageCheck.Start(18);
```

Running this, you can see there's no measurable difference, but the age calculation was done on another thread.

That concludes our tutorial. I hope you've learned something new if you already had experience, and I wish you the best of luck in programming if you're a newbie or not. Please leave some feedback if you enjoyed it or spotted a mistake. :)

Mr Kewl - February 19, 2016, 12:24 pm

Can I interrupt for no reason?

nique ta mere - February 19, 2016, 12:25 pm

Very detailed tutorial, thanks.

Oliver Sykes - February 19, 2016, 3:41 pm

Nice tutorial. Learned some new things about properties and unsafe code/pointers. Thanks for the high quality contribution Odis. Will you be expanding/following up on this?

shiapouf - February 19, 2016, 4:40 pm

Nice Thread sir! It really helps me to understand the basics fundamentals of c#, anyways will you teach me/us about network programming in c#.. coz im planning to create my first project as c# newbie a LAN CHAT like skype or yahoo messenger, with a twist. Thanks. If there is some kind hearted that teach me and guide me in programming c# i deeply appreciate it. thanks a lot

MikaDev - February 19, 2016, 5:03 pm

Holy shit Odis, that's some really good stuff right there.. There's still things in here I can learn from, like the unsafe stuff :) Thanks for taking the time to make this!

Packets395 - February 19, 2016, 5:13 pm

Looks like a nice guide for beginners :) great job Odis.

Algorithms - February 19, 2016, 8:11 pm

Good guide, nice read.

Thanks for sharing, I'll make sure to recommend this to all the new guys looking to learn C#. :')

GuRuHaX - February 20, 2016, 9:06 am

Bookmarked this !

Greatly put together ?

?

KillaMuvz_temp - February 20, 2016, 9:54 am

Very good job on this tutorial, you make it extremely simple.

Ramsey - February 20, 2016, 10:19 am

Thank you so much for this man!

I honestly really appreciate this!

Only3 - February 20, 2016, 11:38 am

? 2 stars rating ... no wonder

AnonCensored - February 20, 2016, 12:16 pm

(Feb 20, 2016 12:38 PM)Only3 Wrote: [\[CITE\]](#) ? 2 stars rating ... no wonder

The fuck does rating have to do with anything ?

Only3 - February 20, 2016, 12:26 pm

(Feb 20, 2016 01:16 PM)AnonCensored Wrote: [\[CITE\]](#) The fuck does rating have to do with anything ?

jealous kids who else rate detailed coding guide with 1 or 2 stars

Khamul - February 20, 2016, 4:01 pm

(Feb 20, 2016 01:26 PM)Only3 Wrote: [\[CITE\]](#) jealous kids who else rate detailed coding guide with 1 or 2 stars

It was rated 1 star but then I 5 starred it. This should be stickied not 1 starred lol

unSatisfied - February 25, 2016, 11:31 am

Great tutorial for beginners, awesome job!

MrJamieMcC - February 29, 2016, 2:28 am

I love this thread! I come here when i have a quick question. Thank you Odus! :)

ifiwonthelotto12 - February 29, 2016, 6:21 am

Already finished 50% of it. Really comprehensive and easy to follow, gonna finish it now!

Odus - February 29, 2016, 7:04 am

(Feb 29, 2016 07:21 AM)ifiwonthelotto12 Wrote: [\[CITE\]](#) Already finished 50% of it. Really comprehensive and easy to follow, gonna finish it now!

I'm pleased you and everyone is finding the guide useful, but please don't power through it because it does move at a fast pace, I didn't want to repeat myself in the same chapter. You can of course deviate and build on what's in the guide.

Take your time and if you're learning, awesome!

Oliver Sykes - February 29, 2016, 6:07 pm

Just in case I forgot to say this, this taught me a few things even at an intermediate level, and went into more detail than I'd thought there would be in a huge guide compilation such as this.

Hiiro - February 29, 2016, 6:34 pm

what happened to the old sticky

ifiwonthelotto12 - March 1, 2016, 12:18 am

```
int[] ages = new int[5];
```

cant you make it shorter like:

```
var ages = new int[5];
```

Oliver Sykes - March 1, 2016, 4:19 am

(Mar 1, 2016 01:18 AM)ifiwonthelotto12 Wrote: [\[CITE\]](#) int[] ages = new int[5];

cant you make it shorter like:

```
var ages = new int[5];
```

Usually a preference, but I believe most people do use var with obvious types. Could be wrong.

Odus - March 1, 2016, 4:48 am

(Feb 29, 2016 07:34 PM)Hiiro Wrote: [\[CITE\]](#) what happened to the old sticky

It was old. You said it yourself.

(Mar 1, 2016 01:18 AM)ifiwonthelotto12 Wrote: [\[CITE\]](#) int[] ages = new int[5];

cant you make it shorter like:

```
var ages = new int[5];
```

When initialized on declaration, sure. It's known as implicit and completely optional. Since this is a guide, I wanted to keep everything typed explicitly.

Mr Kewl - March 1, 2016, 4:51 am

Great thread Odus! ?

Hiiro - March 1, 2016, 5:57 am

(Mar 1, 2016 05:48 AM)Odus Wrote: [\[CITE\]](#) It was old. You said it yourself.

When initialized on declaration, sure. It's known as implicit and completely optional. Since this is a guide, I wanted to keep everything typed explicitly. Is all the old information planning to be covered?

Odus - March 1, 2016, 6:09 am

(Mar 1, 2016 06:57 AM)Hiiro Wrote: [\[CITE\]](#) Is all the old information planning to be covered?

Of course. The bulk is already up, I just haven't had time to redo the rest.

Hiiro - March 1, 2016, 6:19 am

(Mar 1, 2016 07:09 AM)Odus Wrote: [\[CITE\]](#) Of course. The bulk is already up, I just haven't had time to redo the rest. Oki doke, looking forward to it.

Ninja - March 1, 2016, 12:50 pm

Great job again, Odus. Lengthy read, but really useful!

MR.Nitro - March 1, 2016, 7:11 pm

Amazing job Odus, Helped me better understand the fundamentals of c# .

Josh? - March 4, 2016, 8:51 am

I PMed you 3 typos I found, sorry for picking holes in your tutorial.

Other than those, detailed and helped me. Thanks man ?

Odus - March 4, 2016, 9:51 am

(Mar 4, 2016 09:51 AM)Josh? Wrote: [\[CITE\]](#) I PMed you 3 typos I found, sorry for picking holes in your tutorial.

Other than those, detailed and helped me. Thanks man ?

Oops that's my bad, thanks for picking up on those. May need to go over it again if little things like this are present, don't want to confuse people learning.

Glad everyone likes it! I'm writing up a chapter on threading tonight.

B FOR BADASS - March 25, 2016, 5:45 pm

Such clean thread. How much time spent to build one.

FuRnZ - April 20, 2016, 1:10 am

Thanks heaps mate.

The guide's in depth and I seem to be learning a lot more going through this than some eBooks.

wong - April 30, 2016, 10:40 am

Very nice, helped me a lot, great mann thanks

Mr.Motivation - May 1, 2016, 9:26 am

awesome thread you have made here!

Captain Steve Rogers - May 2, 2016, 7:07 am

Thanks for the great tutorial, very helpful for beginners!

Yoshi - May 3, 2016, 2:45 am

Very detailed guide, thanks for the great effort! Can't wait for the rest ?

LittleHorn - May 5, 2016, 2:39 pm

Great write up there, you have a really understandable way of writing - if you wrote a full book I would buy it.

Do you have any plans to extend this further? I'm particularly interested in dependency injection and interfaces.

Thanks again!

Milkcarton - May 8, 2016, 10:26 am

Fuck that is a nice guide for beginners. Would make an awesome eBook ;)

Oliver Sykes - May 8, 2016, 5:18 pm

Still waiting for Threading section, Odis.

Hamzah - May 22, 2016, 7:44 pm

Thank you for the tutorial

Charles Carmichael - May 27, 2016, 8:39 am

Thanks for this looks very solid and posting to come back to this for future reference if need be :)

Oliver Sykes - June 4, 2016, 5:48 am

So, as far as I know, Odus has put this at the end of his priorities, so if you're expecting an update, don't hold your breath.

Wakka - June 18, 2016, 9:34 am

excellent job on this one OP.

Younis Khan - August 11, 2016, 12:34 pm

edit:i posted this on the wrong thread...
fuck my life.

Oliver Sykes - August 11, 2016, 6:44 pm

rip thread rip odus rip isis rip lambda god

Younis Khan - August 11, 2016, 8:16 pm

I've gotten through 2 posts of this guide. Can anyone tell me if I'm doing this right? It works, but is there anything I should take note of or anything I could have done better?

I combined some of the code in the first lesson with some of the code in the second lesson.

```
Code using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;
```

```
namespace CSConsole  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Welcome! Please enter your name.");  
            string name = Console.ReadLine();  
            // nxxtmsg is used to add a period  
            const string nxxtMsg = ".";  
            const string welcomeMsg = "Welcome, ";  
            // this means "Welcome, " plus whatever the user has put in, plus a period.  
            Console.WriteLine(welcomeMsg + name + nxxtMsg);  
            // "Let's begin," then name, then period.  
            const string nextMsg = "Let's begin, ";  
            Console.WriteLine(nextMsg + name + nxxtMsg);  
            // width input  
            Console.Write("Enter the width: ");  
            string input = Console.ReadLine();  
            int width = Convert.ToInt32(input);  
            // height input  
            Console.Write("Enter the height: ");  
            input = Console.ReadLine();  
            int height = Convert.ToInt32(input);  
            // calculate it  
            int perim = (width * 2) + (height * 2);  
            // from here to the next commentary  
            const string newMsg = ", the peremiter of ";  
            const string andMsg = " and ";  
            const string isMsg = " is ";
```

```

        Console.WriteLine(name + newMsg + height + andMsg + width + isMsg + perim + nxxtMsg);
        // is all used to say "Name, the peremiter of HEIGHT and WIDTH is PERIM."
        Console.WriteLine("Press the Enter key when you'd like to exit the program.");
        Console.ReadLine();
    }
}
}

```

jolovic - September 27, 2016, 2:50 am

Wow , this is nice tutorial since I'm starting to learn c#.Thank you.

devfrosso - November 6, 2016, 1:38 pm

Excellent work!! Very helpful! Thanks ?

Mr. Sultan - January 10, 2017, 1:05 am

Amazing! However, almost all the image links are broken.

Malum™ - January 10, 2017, 1:43 am

(Aug 11, 2016 08:16 PM)Maaz Wrote: [CITE](#) I've gotten through 2 posts of this guide. Can anyone tell me if I'm doing this right? It works, but is there anything I should take note of or anything I could have done better?

I combined some of the code in the first lesson with some of the code in the second lesson.
Here you go.

```

Code using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

```

```

namespace CSConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Welcome! Please enter your name: ");
            string name = Console.ReadLine();

            Console.WriteLine($"Welcome, {name}. Let's begin!");

            Console.Write("Enter the width: ");
            string width = Console.ReadLine();

            if (!IsNumeric(width))
            {
                Console.WriteLine($"'{width}' is not a number!");
                Console.ReadLine();
                Environment.Exit(-1);
            }

            int _width = Convert.ToInt32(width);

            Console.Write("Enter the height: ");

```

```

string height = Console.ReadLine();

if (!IsNumeric(height))
{
    Console.WriteLine($"'{height}' is not a number!");
    Console.ReadLine();
    Environment.Exit(-1);
}

int _height = Convert.ToInt32(height);

int perimeter = (_width * 2) + (_height * 2);

Console.WriteLine($"{name}, the perimeter of '{_width}' and '{_height}' is {perimeter}.\nPress
any key to exit.");
Console.ReadKey();
}

public static bool IsNumeric(string input)
{
    int test;
    return int.TryParse(input, out test);
}
}
}

```

JackRed - February 2, 2017, 2:15 pm

Good work - complete and useful! Thank you! ?

Dolippo - March 11, 2017, 3:59 am

Thank's a lot for all this resources, this will be helpfull for undesrtanding more things about C#

T3Xi - August 20, 2017, 8:19 am

Thank you for the effort man! I appreciate that <3.

MNETA - May 28, 2018, 1:41 pm

Absolutely amazing tutorial, I need C# for my computer science course soon, going to start learning with this. Great job!

boffin_mixer - June 25, 2018, 9:53 pm

This is awesome man! Thank you! Bookmarked!

Mawin - July 7, 2018, 4:14 am

Thank you for this guide. This information is very helpful.

Neta - July 23, 2018, 2:14 pm

Amazing guide, bookmarked!

mangolandia - July 23, 2018, 4:48 pm

This was the bomb, but I couldn't finish it, I don't know how to set a value for a "read only" property through the constructor because the image(s) are missing. It said "Property or indexer 'CSCConsole.Student.Age' cannot be assigned to -- it is read only" whenI tried it this way:

```
" public Student(int age)
{
    this.Age = age;
}
"
```

Would be great if the images were updated/fixed. Besides that, it was a really great guide, thanks a lot.

Jarymaine - October 2, 2018, 1:54 pm

Thanks for the comprehensive guide bro.

LucidBoost - October 4, 2018, 6:49 am

Thanks for the guide, found it very useful in identifying and fixing a common error I was making.

VIXV - October 21, 2018, 5:47 am

Thanks for the guide man, using this as we speak. Going up from VB.

Strife97 - October 22, 2018, 12:01 am

Thank you so much for this guide. It helps me a lot getting start on learning c#.

Stahcat - October 25, 2018, 11:55 pm

Great guide so far. But when you say the third option, on running VB6, you refer to .NET Core right?

Stahcat - October 31, 2018, 11:57 pm

Still going over this guide daily. It's helped me a lot when my teacher doesn't.

Drylotrans - January 8, 2019, 5:13 pm

Nice, thanks for the guide, helped me a lot

Dr. Gaster - January 13, 2019, 10:05 am

Thank you so much for your structured guide, it really helped me to understand C# language.

Studio Byte - March 8, 2019, 7:01 am

Thanks , very helpful for beginners.

Effexion - March 8, 2019, 7:05 am

This isn't comprehensive. People just use words that seem to fit these days lol.

Aristocles - March 8, 2019, 9:07 am

(Mar 8, 2019 08:05 AM)Effexion Wrote: [\[CITE\]](#) This isn't comprehensive. People just use words that seem to fit these days lol.

02-19-2016, 09:22 PM (This post was last modified: 03-05-2016, 10:28 AM by Odus.)

This isn't from "these days". People just use words that seem to fit these days lol.

Also, very constructive comment on such a thread.

Effexion - March 8, 2019, 1:14 pm

(Mar 8, 2019 10:07 AM)Aristocles Wrote: [\[CITE\]](#) 02-19-2016, 09:22 PM (This post was last modified: 03-05-2016, 10:28 AM by Odus.)

This isn't from "these days". People just use words that seem to fit these days lol.

Also, very constructive comment on such a thread.
It is constructive as it was pointing out a massive flaw in the title.

Metrole - May 25, 2019, 3:15 pm

I appreciate the work put into the thread!

It's only the pictures which miss in the tutorial that make it a little bit heavier to code and understand.

omar18881 - July 27, 2019, 6:22 pm

Keep sharing Keep learning , I appreciate your work sire Thank you

SniperMait - August 19, 2019, 2:00 pm

thank you, good share , very informing!

Araz5 - May 9, 2020, 8:38 am

thanks so much for the information

Ajay224 - August 29, 2020, 7:11 am

Feel better I guess it was amazing and beautiful concept

Poet - August 29, 2020, 5:11 pm

I guess i just learned basic C#, great... now i need to practice.

BaldCunt - January 7, 2021, 1:59 am

Thank you.

Helped me alot with my current HWs.

Reaqy1 - February 4, 2021, 3:16 am

Sad that some images are no longer available

versx - August 16, 2021, 8:50 pm

Great basic tutorials and explanation for beginners.

Louis Vuitton - October 7, 2021, 4:53 pm

Holy Toledo nice share good shiz

bublaku - April 24, 2022, 6:04 am

Thanks, very detailed tutorial!

windowmaker002 - October 1, 2022, 11:15 am

This is really helpful, thanks alot!

notme001 - October 12, 2022, 8:13 am

i am new her help me to improve my skills

i am new her help me to improve my skills