Part1 – C# Introduction

Introduction – What is c#

- The C# programming language is used for many kinds of applications, including websites, cloud-based systems, IoT devices, machine learning, desktop applications, embedded controllers, mobile apps, games, and command-line utilities
- .NET is cross-platform and open source
- .Net framework and .Net Core (.Net)
- Why c#: range of programming techniques it supports. For example, it offers object-oriented features, generics, and functional programming. It supports both dynamic and static typing. . It provides powerful list- and set-oriented features, thanks to Language Integrated Query (LINQ). It has intrinsic support for asynchronous programming. C# provides options for balancing ease of development against performance. built-in class libraries, and extensive third-party library support.

in 2014, the .NET Foundation was created to foster the development of open source projects in the .NET world. The runtime has always provided a garbage collector (GC) that frees developers from much of the work associated with recovering memory that the program is no longer using.

CLR – Common Language Runtime

- It supports not just C# but any
- The CLR has a Common Type System (CTS) that enables code from multiple languages to interoperate freely, which means that .NET libraries can normally be used from any .NET language—F# can consume libraries written in C#, C# can use Visual Basic libraries, and so on.
- There is an extensive set of class libraries built into .NET. These have gone by a few names over the years, including Base Class Library (BCL), Framework Class Library, and framework libraries, but Microsoft now seems to have settled on runtime libraries as the name for this part of .NET.

.NET language. Microsoft also offers Visual Basic, F#, and .NET extensions for C++, for example.

For years, the most common way for a compiler to work was to process source code and to produce output in a form that could be executed directly by the computer's CPU.

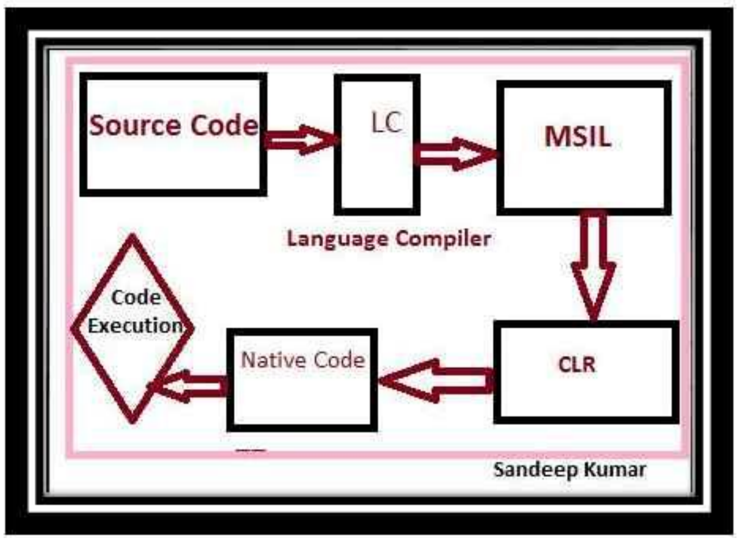
Compilers would produce machine code—a series of instructions in whatever binary format was required by the kind of CPU the computer had. Many compilers still work this way, but the C# compiler does not. Instead, it uses a model called managed code.

Managed Code

- With managed code, the compiler does not generate the machine code that the CPU executes.
- CLR uses an approach called just-in-time (JIT) compilation, in which each individual function’s machine code is generated the first time it runs.

Common Language Runtime

- CLI: Common Language Infrastructure (CLI)
- CLR is responsible for providing common syntax, automatic memory management, and common data type. Also, the responsibility of CLR is generating native code from MSIL Code.
- CLS (Common Language Specification): each and every language in dotnet has their own syntax for writing code and one language does not support the syntax of another language. That means .NET has features of common syntax which are supported by all 63 programming languages.
- CTS (Common Type System) : each and every language int dotnet has their own data type system and One language does not support data type of another language. That means .NET has features of Common Data type which is supported by all 63 programming languages.



BCL: Base class library also known as a Class Library (CL). **BCL** is a subset of the Framework class library (FCL)

***CTS (Common Type System)* Example.**

int i = 123; // There is integer data type in C# .NET

Dim I As Integer // There is Integer data type in J# .NET

As we are looking both languages have their own data type system. But these languages will support the common data type of .NET which is: Int 32

Common Type System

This provides guidelines for declaring, using and managing data types at runtime. It offers cross-language communication. For example, VB.NET has an integer data type, and C# has an int data type for managing integers. After compilation, Int32 is used by both data types. So, CTS provides the data types using managed code. A common type system helps in writing language-independent code.

Common type system provides two categories of types

1. **Value Type:** A value type stores the data in memory allocated on the stack or inline in a structure. This category of Type holds the data directory. If one variable's value is copied to another, both variables store data independently. It can be of inbuilt-in types, user-defined, or enumerations types. Built-in types are primitive data types like numeric, Boolean, char, and date. Users in the source code create user-defined types. An enumeration refers to a set of enumerated values represented by labels but stored as a numeric type.
2. **Reference Type:** A Reference type stores a reference to the value of a memory address and is allocated on the heap. Heap memory is used for dynamic memory allocation. Reference Type does not hold actual data directly but holds the address of data. Whenever a reference-type object is made, it copies the address and not the actual data. Therefore two variables will refer to the same data. If the data of one Reference Type object is changed, the same is reflected for the other object. Reference types can be self-describing types, pointer types, or interference types. The self-describing types may be string, array, and class types that store metadata about themselves.

Common Language Specification

Common Language Specification (CLS) contains a set of rules to be followed by all NET-supported languages. The common rules make it easy to implement language integration and help in cross-language inheritance and debugging. Each language supported by NET Framework has its own syntax rules. But CLS ensures interoperability among applications developed using NET languages.

Assemblies

An assembly is a fundamental unit of physical code grouping. It consists of the assembly manifest, metadata, MSIL code, and a set of resources like image files. It is also considered a basic deployment unit, version control, reuse, security permissions, etc.

Components of the .NET Runtime

- ***Garbage Collector*** is responsible for automatic memory management of objects. Also, the responsibility of the garbage collector is removing the object from memory that has no use.
- ***JIT Compiler*** is responsible for compiling the MSIL Code and generate the Native Code. There are the following 3 types of JIT compiler,
    - Normol Jitter: Normal jitter is a Default jitter that compiles the code line by line and it stores the cache of compiled code.
    - Echono Jitter: Echono jitter compiles the selected line of code and it does not contain the cache of compiled code.
    - Pre Jitter: Pre Jitter Compile the code from top to button and also it stores the cache of compiled code.

Garbage Collector

In Garbage Collector Memory divided into 3 generation:

- When a new object creates it gets stored in Generation 0 and the process gets repeated till generation 0 is full.When Generation 0 size gets full then there 2 processes will be performed.
    - First Process is marking the object which is not reachable for a long time.
    - And in the Second process object will remove from the memory that has marked early and the remaining object will get transferred to Generation 1.
- This process will repeat also for Generation 2. There are 3 methods for removing objects from Memory. These methods are (1) Dispose () (2) Finalize () (3) gc.Collect ().
    - **Dispose Method**: Dispose method is used to remove the object from memory that has no reference this method gets called by the User.
    - **Finalize Method**: Finalize method is used to clean up the object from memory. This method gets called by System.
    - **Gc.Collect Method**: This method is used for removing the object from memory forcefully.

GC Steps

- **1- Suspension:** It suspends the execution of your ENTIRE application. your application is frozen completely when the GC execution starts.
- **2- Sweeping/Marking:** The GC starts scanning the heap looking for objects that have lost their references and, therefore, will no longer be needed by your application and starts removing them.

- **3-Shrinking:** At this step, the GC starts removing all the spaces, moving all these objects in memory (and this is very expensive but necessary) so that the spaces that were in the middle are moved after the last allocated object, and allocation of new objects can be sequential.
- **4-Resume:** Finally our application unfreezes and things begin to run again.

Reference Type and Value Type

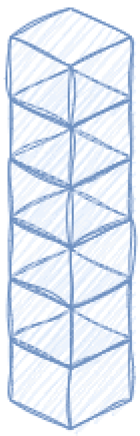
- The Stack is the part of memory which in essence keeps track of everything that is executed (ran) in our code, and value types.
- Heap is responsible for keep track of our objects and structures (reference types).
- Rules:
    - 1. A Reference Type always goes on the Heap
    - 2. Value Types always go on the Stack - Unless, they're declared within a Reference Type, then they go on the Heap.
    - 3. Pointers to the reference type are stored on the Stack.
    - 4. Global variables will be stored on the Heap as they need to be available to everything.

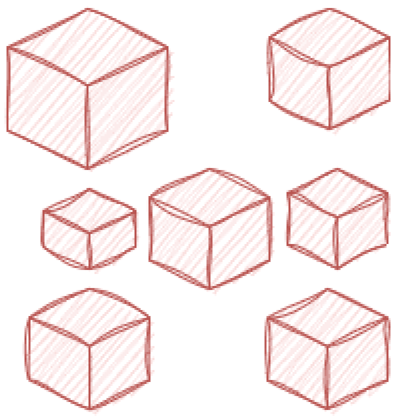
Value types	Reference types
• **Bool**	• class
• **Byte**	• interface
• **char**	• delegate
• **decimal**	• object
• **double**	• string
• **enum**	
• **float**	
• **int**	
• **long**	
• **sbyte**	
• **short**	

• struct **• uint** **• ulong** **• ushort**	

Heap and Stack

Stack memory	**Heap memory**
Stores temporary data by functions and procedures	Stores data not tied to a specific function or procedure
LIFO data structure	No specific structure
Automatic allocation and deallocation by the program	Dynamic allocation and deallocation during runtime
Limited, fixed size	Dynamically adjusted size
Easy to manage	More challenging to manage
Faster access	Slower access
Used for small amounts of data	Used for large, comlex data structures


Stack Structure


Heap Structure

Anatomy of a Simple Program

- Install Dotnet Sdk 6+
- Extenstion: C# Dev Kit
- Dotnet CLI: command-line interface
- dotnet new console
- top-level statements
    - int x = 12 * 30;
    - Console.WriteLine (x);
- dotnet run
- Without top-level statements, a simple console or Windows application looks like this:

```
using System;
class Program
{
        static void Main() // Program entry point
        {
                int x = 12 * 30;
                Console.WriteLine (x);
        }
}
```

Namespaces

- Most .NET types are defined in a namespace. There are certain conventions for namespaces that you'll see a lot. For example, types in .NET's runtime libraries are in namespaces that start with System.
- Using
    - Resolving ambiguity with aliases
    - using IoPath = System.IO.Path;
    - using WpfPath = System.Windows.Shapes.Path;
- Nested namespaces

The .NET runtime libraries contain a large number of types, and there are many more out there in third-party libraries, not to mention the classes you will write yourself. There are two problems that can occur when dealing with this many named entities. First, it becomes hard to guarantee uniqueness. Second, it can become challenging to discover the API you need; unless you know or can guess the right name, it's difficult to find what you need from an unstructured list of tens of thousands of things. Namespaces solve both of these problems.

As you've already seen, the .NET runtime libraries nest their namespaces, sometimes quite extensively, and you will often want to do the same. There are two ways you can do this.

Part 2 - Basic Coding in C#

Local Variables

- C# is a statically typed language, which is to say that any element of code that represents or produces information, such as a variable or an expression, has a data type determined at compile time. This is different than dynamically typed languages, such as JavaScript, in which types are determined at runtime.

```
string part1 = "the ultimate question";
string part2 = "of something";
int theAnswer = 42;
int andAnotherThing;
```

- [datatype] [variableName]=[initialValue];
- var [variableName]=[initialValue];

```
var theAnswer = 42;
theAnswer = "The compiler will reject this";
```

- A variable declared with var behaves in exactly the same way as the equivalent explicitly typed declaration
- Multiple variables in a single declaration
- Keyword default: initialize a variable of any type: value of zero, false, or null.

```
double a, b = 2.5, c = -3;
```

1- The name must begin with either a letter or an underscore, which can be followed by any combination of letters, decimal digits, and underscores. (At least, those are the options if you stick to ASCII.

2- These same rules determine what constitutes a legal identifier for any user-defined entity in C#, such as a class or a method.

3- You can let the compiler work it out for you by using the keyword var in place of the data type. var does not work the same way in C# as in JavaScript: these variables are still all statically typed. All that's changed is that we haven't said what the type is—we're letting the compiler deduce it for us. It looks at the initializers and can see that the first two variables are strings, whereas the third is an integer.

If you try to use the var keyword without an initializer, you'll get a compiler error.

4 – Var may cause to reduce readability for next code review. So it recommended to use var in two situations not for all:

a) when you are repeating type in two sides: List<int> data=new List<int>();

b) when you are using anonymous types (In fact, the var keyword was introduced to C# only when anonymous types were added.)

Scope

- A variable's scope is the range of code in which you can refer to that variable by its name.
- a variable's scope starts at its declaration and finishes at the end of its containing block.
- A block is a region of code delimited by a pair of braces ({}).
- a variable defined in one method is not visible in a separate method, because it is out of scope.

```
static void SomeMethod()
{
    int thisWillNotWork = 42;
}

static void AnUncompilableMethod()
{
    Console.WriteLine(thisWillNotWork);
}
```

- where a nested block starts, everything that is in scope in the outer block continues to be in scope inside that nested block.

```
int someValue = GetValue();
if (someValue > 100)
{
    Console.WriteLine(someValue);
}
```

Variables are not the only things with scope. Methods, properties, types, and, in fact, anything with a name all have scope.

Variable Name Ambiguity

- C# tries to prevent ambiguity by disallowing code where one name might refer to more than one thing in scope.

```
int someValue = GetValue();
if (someValue > 100)
{
    int anotherValue = someValue - 100;  // Compiler error
    Console.WriteLine(anotherValue);
}

int anotherValue = 123;
```

Statements

- When we write a C# method, we are writing a sequence of statements.
- It might be tempting to think of a statement as an instruction to do one thing (such as initializing a variable or invoking a method).
- Or where anything ending in a semicolon is a statement.
- Some categories of statements: declaration, iteration,jump, try/catch/finally, , selection, expression, …

```
int a = 19;
int b = 23;
int c;
c = a + b;
Console.WriteLine(c);
```

it’s the semicolons that are significant here, not the line breaks, by the way.

The first three lines of Example are declaration statements, statements that declare and optionally initialize a variable. The fourth and fifth lines are expression

Statements

When you write a loop, that’s an iteration statement. When you use the if or switch mechanisms to choose between various possible actions, those are selection statements.

Expressions

- a sequence of operators and operands.
- The simplest expressions are literals, where we just write the value we want
- You can use the name of a variable as an expression.
- Expressions can involve operators, which describe calculations or other computations to be performed.
- Operators have some fixed number of inputs, called operands.
- Some take a single operand, such as +.
- Operators also have evaluation order. Ex: (2+2*3/4^2)/(4+2*0)

The fixed values are called as ***Literal***. Literal is a value that is used by the variables.

// Here 100 is a constant/literal.
int x = 100;

Comments and Whitespace

- allow source files to contain text that is ignored by the compiler. two styles of comments: single-line (//), delimited comments (/* */)

```
Console.WriteLine("Say");        // This text will be ignored, but the
code on

Console.WriteLine("This will run");   /* This comment includes not
just the
Console.WriteLine("This won't");       * text on the right but also
the text
Console.WriteLine("Nor will this");   /* on the left except the first
and last
Console.WriteLine("Nor this");         * lines. */
Console.WriteLine("This will also run");
```

- C# ignores extra whitespace for the most part.

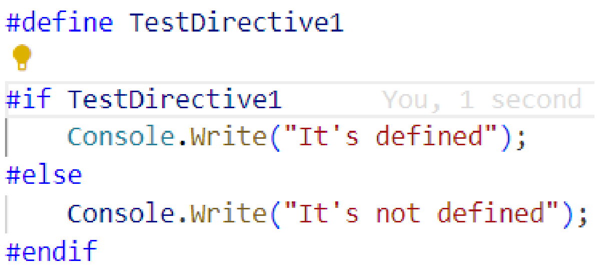
```
Console.WriteLine("Testing");
Console . WriteLine(    "Testing");
Console.
    WriteLine ("Testing" )
  ;
```

Preprocessing Directives

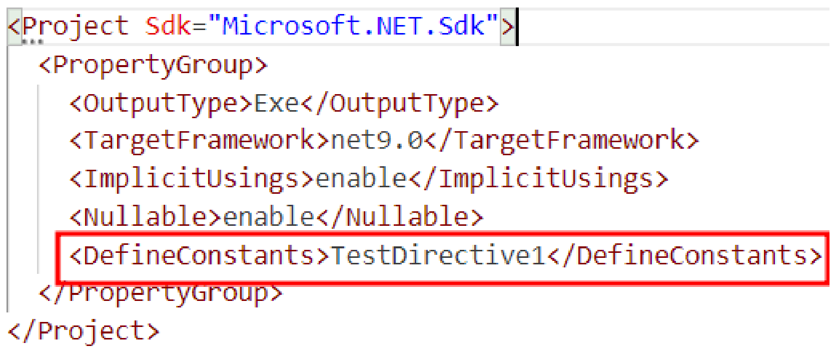
- The preprocessor directives give instruction to the compiler to preprocess the information before actual compilation starts.
- All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line.
- Preprocessor directives are not statements, so they do not end with a semicolon (;).
- In C# the preprocessor directives are used to help in conditional compilation.

Compilation Symbols

- C# offers a #define directive that lets you define a compilation symbol.
- These symbols are commonly used in conjunction with the #if directive to compile code in different ways for different situations.
- You can open up the .csproj file and define the values you want in a <DefineConstants> element of any <PropertyGroup>.



```
#define TestDirective1

#if TestDirective1
    Console.Write("It's defined");
#else
    Console.Write("It's not defined");
#endif
```



```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <DefineConstants>TestDirective1</DefineConstants>
  </PropertyGroup>
</Project>
```

For example, you might want some code to be present only in Debug builds, or perhaps you need to use different code on different platforms to achieve a particular effect.

- ◈ The .NET SDK defines certain symbols by default. It supports two configurations, Debug and Release. It defines a DEBUG compilation symbol in the Debug configuration, whereas Release will define RELEASE instead.

- ◈ conditional method: indicate that the annotated method should be used only when the DEBUG compilation symbol is defined.

```
[System.Diagnostics.Conditional("DEBUG")]
static void ShowDebugInfo(object o)
{
    Console.WriteLine(o);
}
```

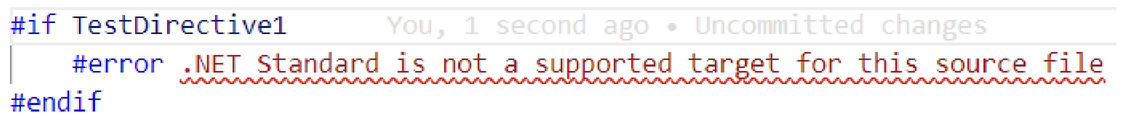
If you write code that calls a method that has been annotated in this way, the C# compiler will omit that call in builds that do not define the relevant symbol.

So if you write code that calls this ShowDebugInfo method, the compiler strips out all those calls in non-Debug builds.

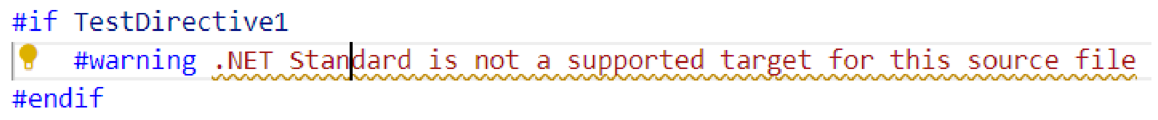
- ◈ #error and #warning: generate compiler errors or warnings.

- ◈ These are typically used inside conditional regions.

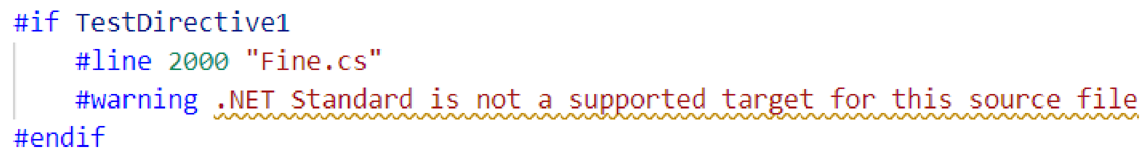
- ◈ #line: If want to change some info for error and warning.



```
#if TestDirective1        You, 1 second ago • Uncommitted changes
    #error .NET Standard is not a supported target for this source file
#endif
```



```
#if TestDirective1
    #warning .NET Standard is not a supported target for this source file
#endif
```



```
#if TestDirective1
    #line 2000 "Fine.cs"
    #warning .NET Standard is not a supported target for this source file
#endif
```

When the compiler produces an error or a warning, it states where the problem occurred, providing the filename, a line number, and an offset within that line.

- ◈ #pragma provides two features:
    - ◇ it can be used to disable selected compiler warnings,
    - ◇ You should generally avoid disabling warnings.
    - ◇ be used to override the checksum values the compiler puts into the .pdb file it generates containing debug information.

- #region and #endregion: do nothing. the only thing the compiler does is ensure that they have corresponding #endregion directives.
- These directives exist entirely for the benefit of text editors that choose to recognize them.

Fundamental Data Types

- .NET defines thousands of types in its runtime libraries, and you can write your own, so C# can work with an unlimited number of data types.

Numeric Types

- C# supports integer and floating-point arithmetic. There are signed and unsigned integer types, and they come in various sizes
- The most commonly used integer type is int

C# type	CLR name	Signed	Size in bits	C# type
Byte	System.Byte	No	8	0 to 255
Sbyte	System.Sbyte	Yes	8	−128 to 127
ushort	System.UInt16	No	16	0 to 65,535
Short	System.Int16	Yes	16	−32,768 to 32,767
uint	System.UInt32	No	32	0 to 4,294,967,295
int	System.Int32	Yes	32	−2,147,483,648 to 2,147,483,647
ulong	System.UInt64	No	64	0 to 18,446,744,073,709,551,615
long	System.Int64	Yes	64	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
nint	System.IntPtr	Yes	Depends	Depends
nuint	System.UIntPtr	No	Depends	Depends

- C# also supports floating-point numbers.
- Decimal is not just a bigger version of double.

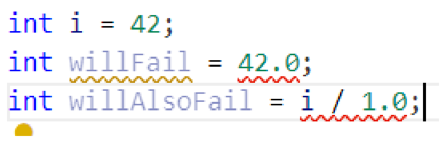
C# type	CLR name	Size in bits	Precision	Range
float	System.Single	32	23 bits (~7 decimal digits)	1.5 * 10^{-45} to 3.4*10^{38}
double	System. Double	64	52 bits (~15 decimal digits)	5.0 * 10^{-324} to 1.7*10^{308}
decimal	System.Decimal	256	28-29 digits	1.0 * 10^{-28} to 7.9228 *10^{28}

Numeric conversions

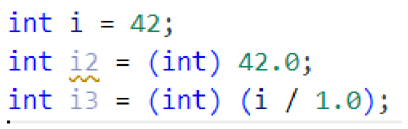
- Each of the built-in numeric types uses a different representation for storing numbers in memory. Converting from one form to another requires some work
- when you perform arithmetic calculations that involve a mixture of numeric types, C# will pick the type with the largest range and promote values of types with a narrower range into that larger one before performing the calculations.

```
int i = 42;
double di = i;
Console.WriteLine(i / 5); //8
Console.WriteLine(di / 5); //8.4
Console.WriteLine(i / 5.0); //8.4

int i1 = 5;
double d1 = i1;
var t1 = i1 / 5;
var t2 = d1 / 5;
var t3 = i1 / 5.0;

Console.WriteLine(t1.GetType());//System.Int32
Console.WriteLine(t2.GetType());//System.Double
Console.WriteLine(t3.GetType());//System.Double
```

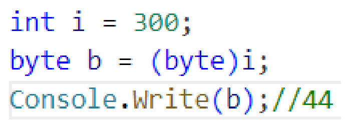
- Implicit casting(automatically) is done automatically when passing a smaller size type to a larger size type: char -> int -> long -> float -> double
- Explicit Casting (manually) - converting a larger type to a smaller size type: double -> float -> long -> int -> char



```
int i = 42;
int willFail = 42.0;
int willAlsoFail = i / 1.0;
```



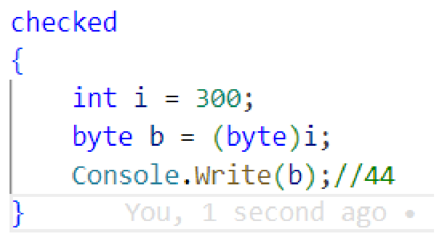
```
int i = 42;
int i2 = (int) 42.0;
int i3 = (int) (i / 1.0);
```



```
int i = 300;
byte b = (byte)i;
Console.Write(b);//44
```

Checked contexts

- This means that certain arithmetic operations, including casts, are checked for range overflow at runtime. If you cast a value to an integer type in a checked context and the value is too high or low to fit, an error will occur—the code will throw a System.OverflowException.



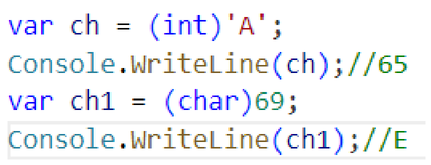
```
checked
{
    int i = 300;
    byte b = (byte)i;
    Console.Write(b);//44
}       You, 1 second ago •
```

Booleans

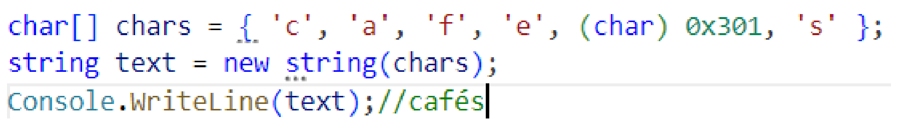
- C# defines a type called bool, or as the runtime calls it, System.Boolean.
- This offers only two values: true and false.

Strings and Characters

- Char type (System.Char) is a 16-bit value representing a single UTF-16 code unit.
- The string type (synonymous with the CLR System.String type) represents text.
- A string is a sequence of values of type char



```
var ch = (int)'A';
Console.WriteLine(ch);//65
var ch1 = (char)69;
Console.WriteLine(ch1);//E
```



```
char[] chars = { 'c', 'a', 'f', 'e', (char) 0x301, 's' };
string text = new string(chars);
Console.WriteLine(text);//cafés
```

Immutability of strings

- .NET strings are immutable.
- They can't be changed after they're created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object.
- If you need to do work that performs a series of modifications to a string, you can use a type called StringBuilder

String manipulation methods

- The string type has numerous instance methods for working with strings.
- ToUpper and ToLower
- IndexOf and LastIndexOf
- StartsWith and EndsWith: return a bool indicating whether the string starts or ends with a particular character or string
- Split: takes one or more separator characters (e.g., commas or spaces) and returns an array with an entry for each substring between the separators
- Substring: remove before starting position and return remained text.
- Insert: forms a new string by inserting one string into the middle of another.
- Replace: returns a new string formed by replacing all instances of a particular character or string with another
- Trim: remove unwanted leading and trailing characters such as whitespace

Formatting data in strings

- syntax that makes it easy to produce strings that contain a mixture of fixed text and information determined at runtime.
- The official name for this feature is string interpolation.

```
string message = $"{name} is {age} years old";
```

```
string message = string.Format("{0} is {1} years old", name, age);
string info = string.Format(
    "Hypotenuse: {0}",
    Math.Sqrt(width * width + height * height));
```

Verbatim string literals

- you can prefix a string literal with the @ symbol like so: @"Hello".
- they can improve the readability of strings containing backslashes, and they make it possible to write multiline string literals.

```
var x1="hello \world";
var x2="hello \\world";
var x3=@"hello world";
```

Dynamic

- C# defines a type called dynamic. This doesn't directly correspond to any CLR type—when we use dynamic in C#, the compiler presents it to the runtime as object.
- With dynamic, the compiler makes no attempt at compile time to check whether operations performed by code are likely to succeed.

```
var x1="hello \world";
var x2="hello \\world";
var x3=@"hello world";
```

Object

- The last data type to get special recognition from the C# compiler is object (or System.Object, as the CLR calls it).
- This is the base class of almost all C# types.
  - There are some specialized exceptions, such as pointer types.

Operators

Name (arithmetic operations)	Example
Unary plus (does nothing)	+x
Negation (unary minus)	-x
Postincrement	x++
Postdecrement	x--
Preincrement	++x
Predecrement	--x

Addition	x + y
Subtraction	x - y
Multiplication	x * y
Division	x / y
Remainder	x % y

Name (binary operations)	**Example**
Bitwise negation	~x
Bitwise AND	x & y
Bitwise OR	x | y
Bitwise XOR	x ^ y
Shift left	x << y
Shift right	x >> y

Name (operators for bool)	**Example**
Logical negation (also known as NOT)	!x
Conditional AND	x && y
Conditional OR	x || y

Name (Relational operators)	**Example**
Less than	x < y
Greater than	x > y
Less than or equal	x <= y
Greater than or equal	x >= y
Equal	x == y
Not equal	x != y

Flow Control

- Conditional:
    - if: decides whether or not to run some particular statement depending on the value of a bool expression
    - switch: Multiple Choice with switch Statements
- Loop:
    - while
    - do-while
    - For
    - Foreach: for Collection Iteration
        - foreach (item-type iteration-variable in collection) body
- break and continue

If

- Single Statement
- Multi Statement (Needs block)
- If else
- If else if

Multiple Choice with switch Statements

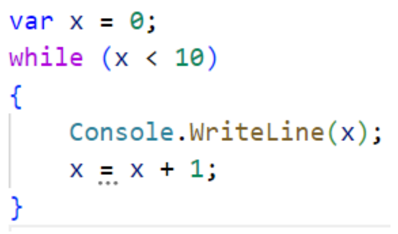
- A switch statement defines multiple groups of statements and either runs one group or does nothing at all, depending on the value of an input expression.
- You can also write a default section, which will run if none of the cases apply.
- break statements, which causes execution to jump to the end of the switch statement.
- This is not the only way to finish a section—strictly speaking, the rule imposed by the C# compiler is that the end point of the statement list for each case must not be reachable, so anything that causes execution to leave the switch statement is acceptable.
- You could use a return statement instead, or throw an exception, or you could even use a goto statement.

```
switch (workStatus)
{
case "ManagerInRoom":
    WorkDiligently();
    break;

case "HaveNonUrgentDeadline":
case "HaveImminentDeadline":
    CheckTwitter();
    CheckEmail();
    CheckTwitter();
    ContemplateGettingOnWithSomeWork();
    CheckTwitter();
    CheckTwitter();
    break;

case "DeadlineOvershot":
    WorkFuriously();
    break;

default:
    CheckTwitter();
    CheckEmail();
    break;
}
```

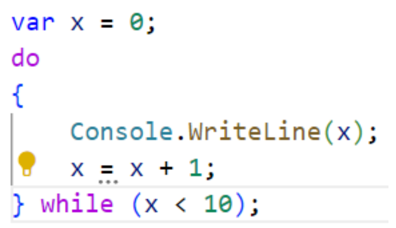
while and do

- This takes a bool expression. It evaluates that expression, and if the result is true, it will execute the statement that follows.
- The body of the loop may decide to finish the loop early with a break statement.
- continue terminates the current iteration, but unlike break, it will then reevaluate the while expression, so iteration may continue.

- continue and break are also available for all of the other loop styles I'm about to show.



```
var x = 0;
while (x < 10)
{
    Console.WriteLine(x);
    x = x + 1;
}
```



```
var x = 0;
do
{
    Console.WriteLine(x);
    x = x + 1;
} while (x < 10);
```

this is just like an if statement, but the difference is that once the loop's embedded statement is complete, it then evaluates the expression again, and if it's true again, it will execute the embedded statement a second time. It will keep doing this until the expression evaluates to false.

As with if statements, the body of the loop does not need to be a block, but it usually is.

It does not matter whether the while expression is true or false—executing a break statement will always terminate the loop.

Both continue and break jump straight to the end of the loop, but you could think of continue as jumping directly to the point just before the loop's closing }, while break jumps to the point just after.

For loop

- for (initializer; condition; iterator) body
- This is similar to while, but it adds two features to that loop's bool expression:
    - it provides a place to declare and/or initialize one or more variables that will remain in scope for as long as the loop runs,
    - it provides a place to perform some operation each time around the loop (in addition to the statement that forms the body of the loop).

Foreach

- Foreach loop is designed for iterating through collections.
- foreach (item-type iteration-variable in collection) body

- The collection is an expression whose type must match a particular pattern recognized by the compiler.
- The runtime libraries' IEnumerable<T> interface matches this pattern.

```
string[] messages = GetMessagesFromSomewhere();
foreach (string message in messages)
{
    Console.WriteLine(message);
}
```

Patterns

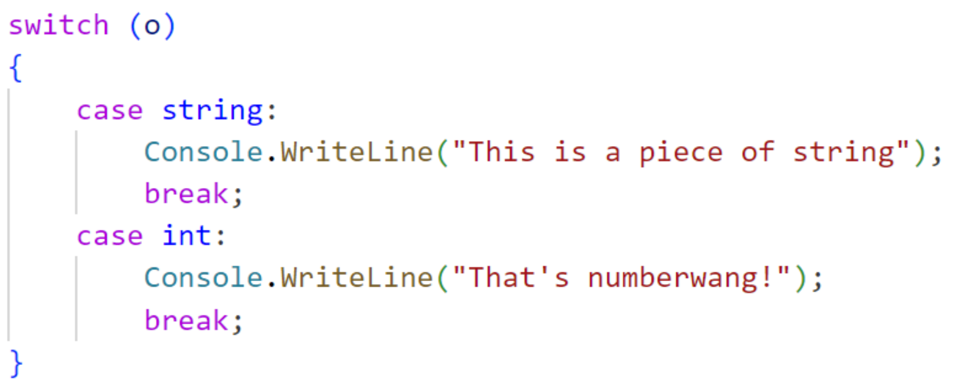
- A pattern describes one or more criteria that a value can be tested against.
- There are many kinds of patterns, and they aren't just for switch statements.
- 1- Constant patterns: With these, you specify just a constant value, and an expression matches this pattern if it has that value.
    - all used one of the simplest pattern types: they were all constant patterns.

```
switch (workStatus)
{
case "ManagerInRoom":
    WorkDiligently();
    break;

case "HaveNonUrgentDeadline":
case "HaveImminentDeadline":
    CheckTwitter();
    CheckEmail();
    CheckTwitter();
    ContemplateGettingOnWithSomeWork();
    CheckTwitter();
    CheckTwitter();
    break;

case "DeadlineOvershot":
    WorkFuriously();
    break;

default:
    CheckTwitter();
    CheckEmail();
    break;
}
```

- 2- Declaration patterns: An expression matches a declaration pattern if it has the specified type.

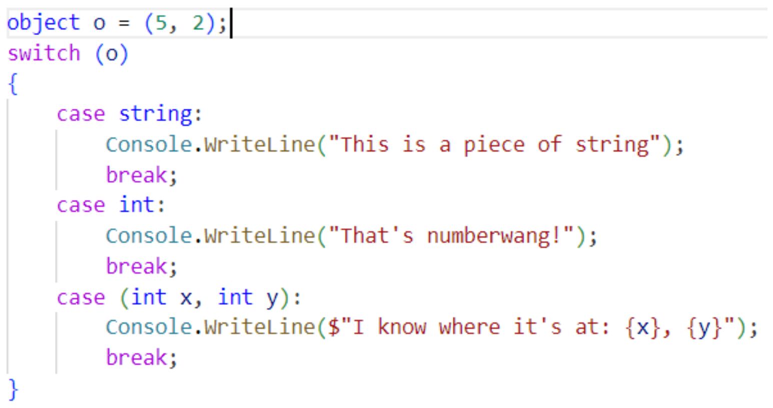
```
switch (o)
{
case string s:
    Console.WriteLine($"A piece of string is {s.Length} long");
    break;

case int i:
    Console.WriteLine($"That's numberwang! {i}");
    break;
}
```

- 3- Type patterns: looks and works like a declaration pattern without the variable



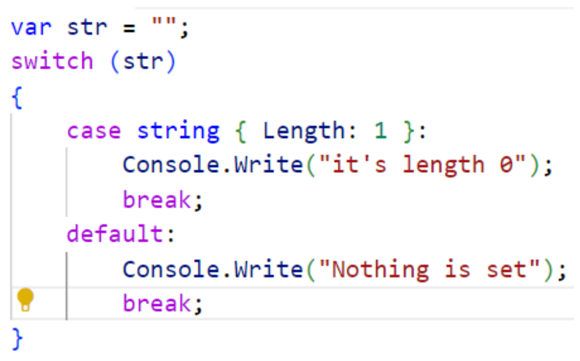
```
switch (o)
{
    case string:
        Console.WriteLine("This is a piece of string");
        break;
    case int:
        Console.WriteLine("That's numberwang!");
        break;
}
```

- 4- positional pattern: matches any tuple containing a pair of int values and extracts those values into two variables, x and y.



```
object o = (5, 2);
switch (o)
{
    case string:
        Console.WriteLine("This is a piece of string");
        break;
    case int:
        Console.WriteLine("That's numberwang!");
        break;
    case (int x, int y):
        Console.WriteLine($"I know where it's at: {x}, {y}");
        break;
}
```

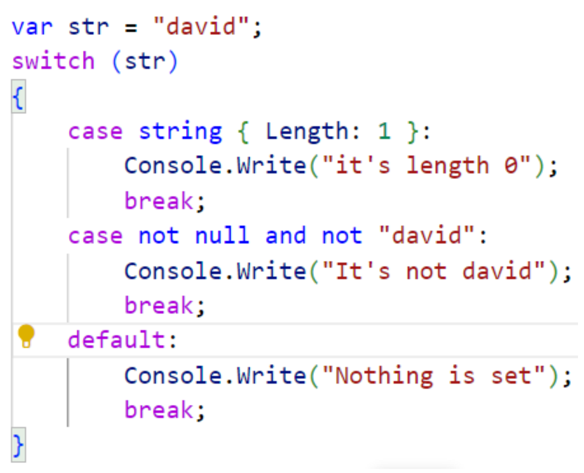
- 5- Property pattern: they are members of a type that provide some sort of information, such as the string type's Length property, which returns an int telling you how many code units the string contains



```csharp
var str = "";
switch (str)
{
    case string { Length: 1 }:
        Console.Write("it's length 0");
        break;
    default:
        Console.Write("Nothing is set");
        break;
}
```

Combining and Negating Patterns

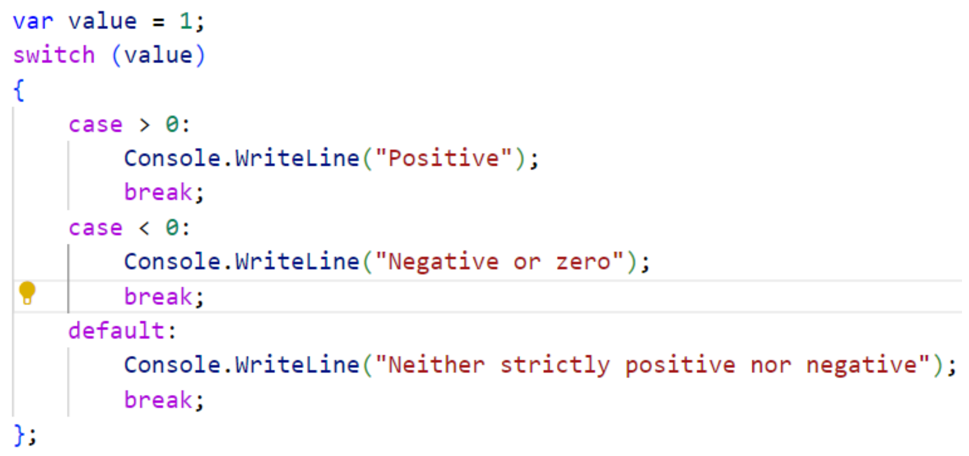
- C# offers three logical operations for use in patterns: and, or, and not.
- Not lets you invert the meaning of a pattern.



```csharp
var str = "david";
switch (str)
{
    case string { Length: 1 }:
        Console.Write("it's length 0");
        break;
    case not null and not "david":
        Console.Write("It's not david");
        break;
    default:
        Console.Write("Nothing is set");
        break;
}
```

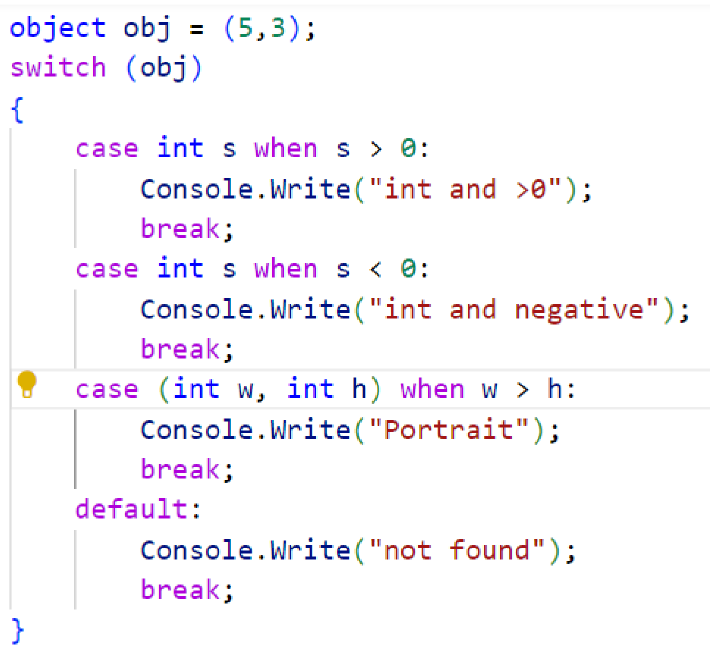
Relational Patterns

- Patterns can use the <, <=, >=, and > operators when the pattern's type supports these kinds of comparison.



```csharp
var value = 1;
switch (value)
{
    case > 0:
        Console.WriteLine("Positive");
        break;
    case < 0:
        Console.WriteLine("Negative or zero");
        break;
    default:
        Console.WriteLine("Neither strictly positive nor negative");
        break;
};
```

Pattern with when clause



```
object obj = (5,3);
switch (obj)
{
    case int s when s > 0:
        Console.Write("int and >0");
        break;
    case int s when s < 0:
        Console.Write("int and negative");
        break;
    case (int w, int h) when w > h:
        Console.Write("Portrait");
        break;
    default:
        Console.Write("not found");
        break;
}
```