C# 7/8/9/10 - Basics

- The logic is contained in a type – just as in java
  - A type is for instance
    - A class
    - An interface
- in C++ uses class definitions in header files and logic in implementation files
- in C# global variables or functions do not exists, or?

```
using System;

class HalloWorld
{
  public static int Main()

  {
    Console.WriteLine("Hallo from C#");
    return 0;
  }
}
```

verdensvigtigsteprogram.cs

- As in C/C++/Java:
  - Every application must have one Main method
  - Are there more Main methods it must be specified which Main to use
- Remember Main – not main
- Can have one single parameter :public static int Main(string[ ] args)
- Comments in source code as in C / C++ / java
- Main can be declared **private**
- Main can return void or an int
  - an int can be used for diagnose

With the release of C# 7.1, the `Main()` method can now be asynchronous. Async programming is covered in Chapter 15, but for now realize there are four additional signatures:

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

Using Error Code

```
// Note we are now returning an int, rather than void.
static int Main(string[] args)
{
  // Display a message and wait for Enter key to be pressed.
  Console.WriteLine("***** My First C# App *****");
  Console.WriteLine("Hello World!");
  Console.WriteLine();
  Console.ReadLine();

  // Return an arbitrary error code.
  return -1;
}
```

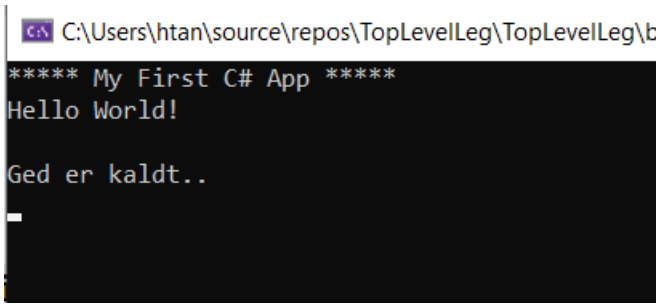
C# 9: Top Level Statements

```
// Display a simple message to the user.
Console.WriteLine("***** My First C# App *****");
Console.WriteLine("Hello World!");
Console.WriteLine();
// Wait for Enter key to be pressed before shutting down.
Console.ReadLine();
```

But you can also.. (WHY???)

```
using System;
// Display a simple message to the user.
Console.WriteLine("***** My First C# App *****");
Console.WriteLine("Hello World!");
Console.WriteLine();

void Ged()
{
    Console.WriteLine("Ged er kaldt..");
}

Ged();
// Wait for Enter key to be pressed before shutting down.
Console.ReadLine();
namespace TopLevelLeg
{
    class Program
    {
```

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



Main is discarded!

This is also possible:

```
using System;
// Display a simple message to the user.
Console.WriteLine("***** My First C# App *****");
Console.WriteLine("Hello World!");
Console.WriteLine();

void Ged()
{
    Console.WriteLine("Ged er kaldt..");
}

Ged();


TopLevelLeg.Program p = new TopLevelLeg.Program();

p.MyMethod();


Console.ReadLine();
namespace TopLevelLeg
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World - from Main!");
        }

        public void MyMethod()
        {
            Console.WriteLine("MyMethod er kaldt..");
        }
    }
```

```
}
```

Specifying the application error code

On the Windows operating system, an application's return value is stored within a system environment variable named `%ERRORLEVEL%`. If you were to create an application that programmatically launches another executable (a topic examined in Chapter 19), you can obtain the value of `%ERRORLEVEL%` using the `ExitCode` property of the launched process.

Now let's capture the return value of the program with the help of a batch file. Using Windows Explorer, navigate to the folder containing your project file (e.g., `C:\SimpleCSharpApp`) and add a new text file (named `SimpleCSharpApp.cmd`) to that folder. Update the contents of the folder to the following (if you have not authored `*.cmd` files before, do not concern yourself with the details):

```
@echo off
rem A batch file for SimpleCSharpApp.exe
rem which captures the app's return value.

dotnet run
@if "%ERRORLEVEL%" == "0" goto success

:fail
  echo This application has failed!
  echo return value = %ERRORLEVEL%
  goto end
:success
  echo This application has succeeded!
  echo return value = %ERRORLEVEL%
  goto end
:end
echo All Done.
```

Commando line arguments and more about WriteLine

```
using System;


public class Hallo{

  private static int Main(string[] args) {

  for(int i = 0;i< args.Length;i++)
   Console.WriteLine("Argument no. {0} {1}",i,args[i]);

   Console.ReadLine();
   return 0;
 }

}
```

ko.cs

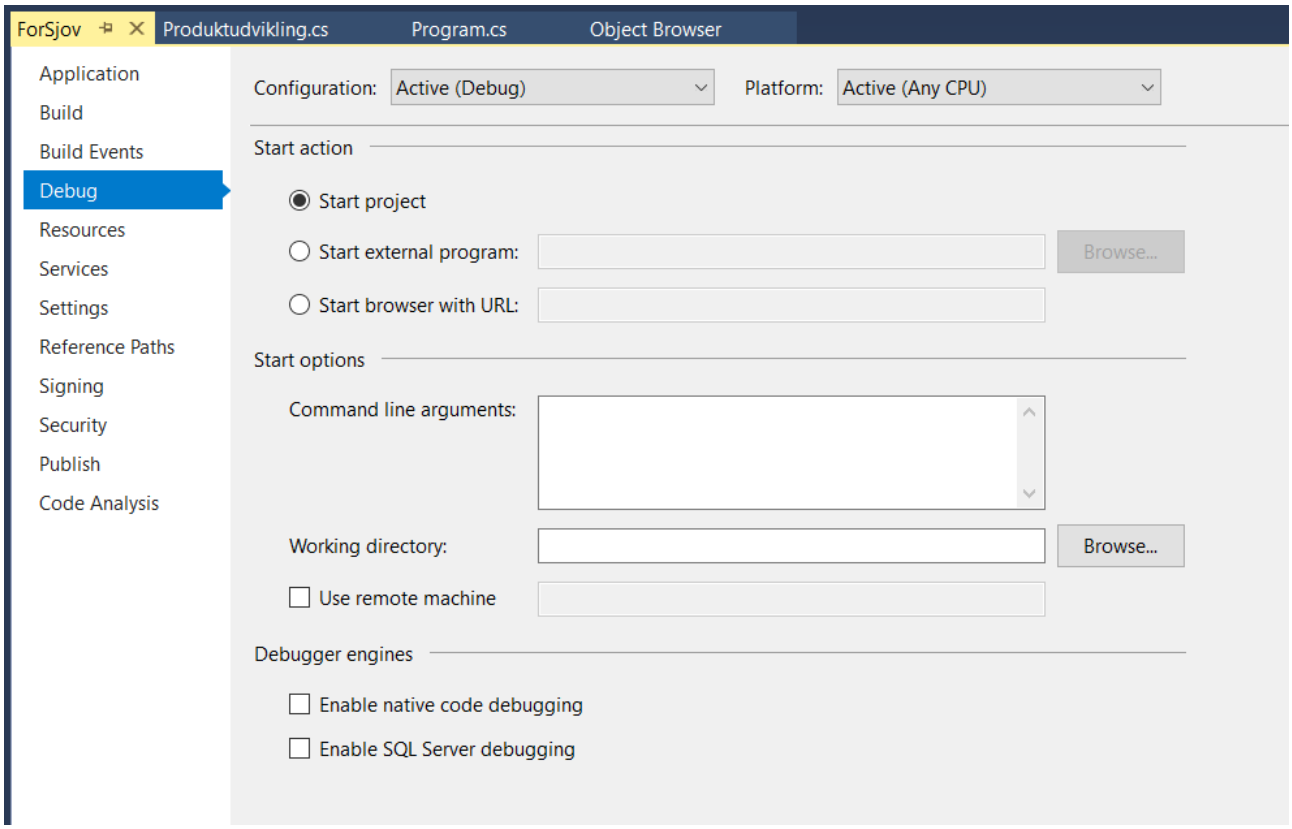
bat file:

```
ko.exe A nice day
```

- gives of course 3 arguments

- Identical with Java – different from C++

Setting Command line args in VS



GetCommandLineArgs()

- Environment type has a method:

```
...
string[] args = Environment.GetCommandLineArgs();
Console.WriteLine("Path:  {0}",args[0]);

  for(int i = 1;i< args.Length;i++)
   Console.WriteLine("Argument no. {0} {1}",i, args[i]);


...
```

- Just as in C++
  - Here: Path is in index 0

C# 9: Command line args on top level

```
…

// Process any incoming args.
for (int i = 0; i < args.Length; i++)
{
Console.WriteLine("Arg: {0}", args[i]);
}
…
```

More about the Environment class

Table 3-1. *Select Properties of System.Environment*

Property	Meaning in Life
ExitCode	Gets or sets the exit code for the application
Is64BitOperatingSystem	Returns a bool to represent if the host machine is running a 64-bit OS
MachineName	Gets the name of the current machine
NewLine	Gets the newline symbol for the current environment
SystemDirectory	Returns the full path to the system directory
UserName	Returns the name of the user that started this application
Version	Returns a Version object that represents the version of the .NET platform

```
static void ShowEnvironmentDetails()
{
  // Print out the drives on this machine,
  // and other interesting details.
  foreach (string drive in Environment.GetLogicalDrives())
  {
    Console.WriteLine("Drive: {0}", drive);
  }
  Console.WriteLine("OS: {0}", Environment.OSVersion);
  Console.WriteLine("Number of processors: {0}",
    Environment.ProcessorCount);
  Console.WriteLine(".NET Core Version: {0}",
    Environment.Version);
}
```

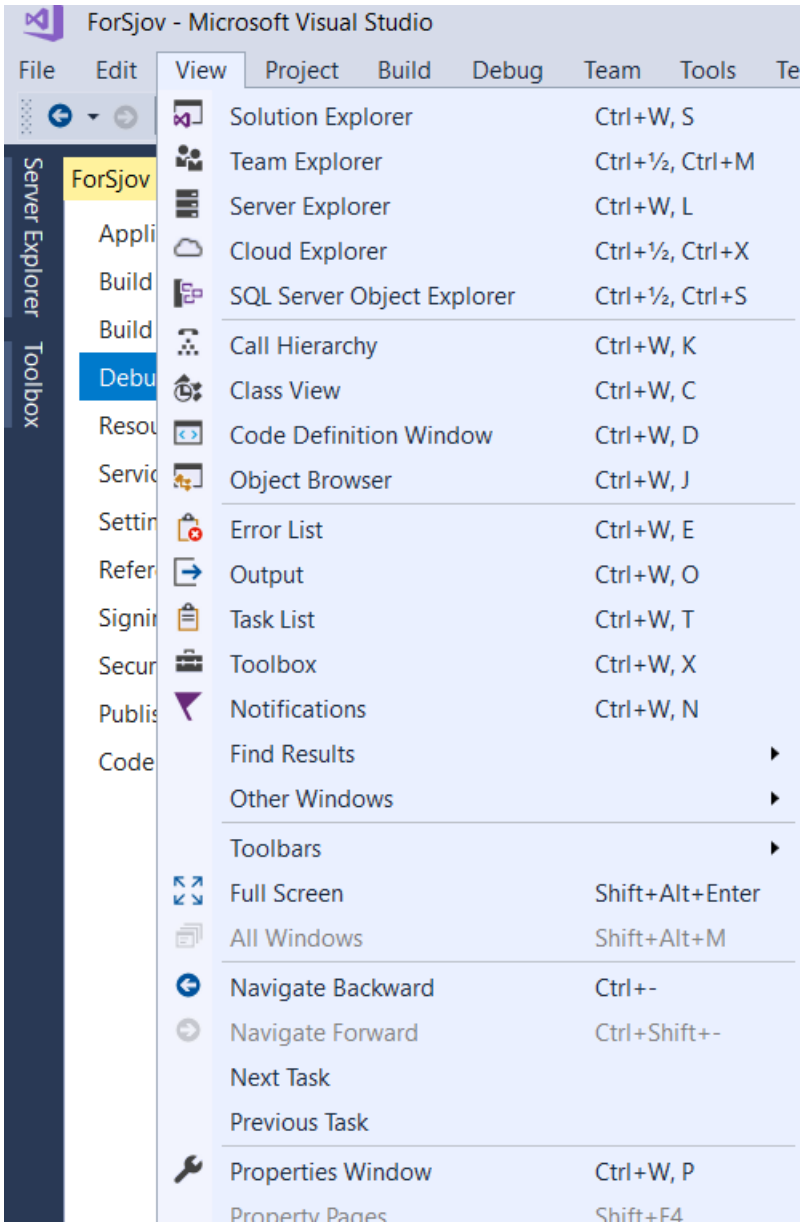
System.Console class

***Table 3-2**. Select Members of System.Console*

Member	Meaning in Life
Beep()	This method forces the console to emit a beep of a specified frequency and duration.
BackgroundColor ForegroundColor	These properties set the background/foreground colors for the current output. They may be assigned any member of the ConsoleColor enumeration.
BufferHeight BufferWidth	These properties control the height/width of the console's buffer area.
Title	This property sets the title of the current console.
WindowHeight WindowWidth WindowTopWindowLeft	These properties control the dimensions of the console in relation to the established buffer.
Clear()	This method clears the established buffer and console display area.

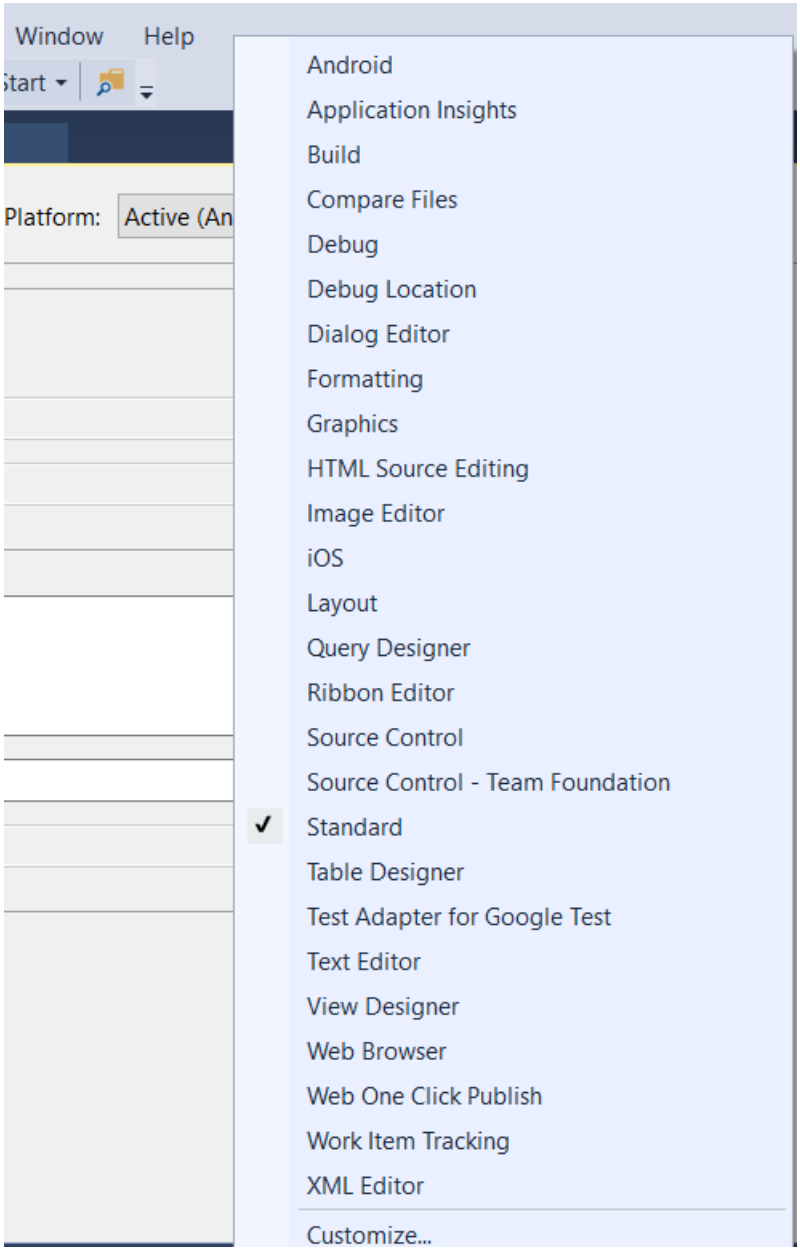
Information to programmers about VS NET

- Handling VS
- Use ”View”



Menu line adjustments

- Right-click



Formatting data

Table 3-3. *.NET Core Numerical Format Characters*

String Format Character	Meaning in Life
C or c	Used to format currency. By default, the flag will prefix the local cultural symbol (a dollar sign [$] for US English).
D or d	Used to format decimal numbers. This flag may also specify the minimum number of digits used to pad the value.
E or e	Used for exponential notation. Casing controls whether the exponential constant is uppercase (E) or lowercase (e).
F or f	Used for fixed-point formatting. This flag may also specify the minimum number of digits used to pad the value.
G or g	Stands for *general*. This character can be used to format a number to fixed or exponential format.
N or n	Used for basic numerical formatting (with commas).
X or x	Used for hexadecimal formatting. If you use an uppercase X, your hex format will also contain uppercase characters.

String.Format

```
// Using string.Format() to format a string literal.
string userMessage = string.Format("100000 in hex is {0:x}", 100000);
```

Local variables and initialization

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // Local variables are declared and initialized as follows:
  // dataType varName = initialValue;
  int myInt = 0;

  // You can also declare and assign on two lines.
  string myString;
  myString = "This is my character data";

  Console.WriteLine();
}
```

The default Literal (New 7.1)

The `default` literal assigns a variable the default value for its data type. This works for standard data types as well as custom classes (Chapter 5) and generic types (Chapter 10). Create a new method named `DefaultDeclarations()` and add the following code:

```
static void DefaultDeclarations()
{
  Console.WriteLine("=> Default Declarations:");
  int myInt = default;
}
```

- `bool` variables are set to `false`.
- Numeric data is set to `0` (or `0.0` in the case of floating-point data types).
- `char` variables are set to a single empty character.
- `BigInteger` variables are set to `0`.
- `DateTime` variables are set to `1/1/0001 12:00:00 AM`.
- Object references (including `strings`) are set to `null`.

Constructors

- An object is an instance of a class
  - Keyword **new** must always be used
  - As in java

- Trying to use an object without calling ”new”:
  - Error on **compile time**

Example:

```
using System;

class HejObjekt
```

```
{
  public static int Main()

  {
    Console.WriteLine("OH – an object);
    AClass anObj = new AClass();
    anObj.MethodCall();

    return 0;
  }
}
```

- new allocates memory on the heap

- As in C++ and Java
  - C# has a default constructor (without arguments)
  - Many constructors can be applied
  - If a selfmade constructor is added – the default constructor is removed

```
using System;


public class Hallo{

 public static int Main()
 {

  EnKlasse etObj      = new EnKlasse();
  EnKlasse etAndetObj = new EnKlasse(25);

  etObj.udskriv();
  etAndetObj.udskriv();
  Console.ReadLine();

  return 0;
 }


}

public class EnKlasse {

  private int y;

  public EnKlasse()
  {
```

```
    y = 0;                  // explicitly:  all member variables are initialized to 0

  }

  public EnKlasse(int i)
 {

     y=i;

  }

  public void udskriv()
 {


   Console.WriteLine("y = {0}",y);
  }

}
```

Garbage Collection

- C# uses automatic Garbage Collection as in java
  - **delete** is not used (C++)

  It is possible to do it manually

Application objects

- Responsibility (OOAD thinking...)

```
using System;


public class StartAction{

  public static int Main()
  {

   Aktion action = new Aktion();
   action.udskriv();
   return 0;
```

```
 }


}

public class Aktion {


    public void udskriv()
   {

    Console.WriteLine("Hi..");

   }

}
```

- StartHallo has a responsibility to:
  - create an Aktion object

Scope of variables

- A member variable is automatically applied a default value (0, null)
  - This is **NOT TRUE** for variables declared in a method
  - Compile-error (where C++ has run-time error)

Example:

```
  public int Sq()
   {
    int tmp;
    return tmp;

   }
```

```
l2_5.cs(22,12): error CS0165: Use of unassigned local variable 'tmp'
```

- but this is (of course) OK (but…)

```
...
   public int Sq(int y)
   {
    int tmp;
    tmp = y*y;
    return tmp;

   }
...
```

Initialization of a member variable

- Instead of using redundant code in constructors
  - Use direct initialization

```
Class Hallo
{
   private int etwas;

  public Hallo()
 {
    etwas = 10;
 }

  public Hallo(string comment)
 {
    etwas = 10;
 }


}
```

- Solution...

```
class Hallo
{
   private int etwas = 10;

  public Hallo()
 {
}

  public Hallo(string comment)
 {

 }


}
```

About Write, Read, WriteLine and ReadLine

- WriteLine and ReadLine uses CR
- Write and Read are not using CR

Formatting in C style

- looks like printf
  - without using the %d flag
  - Based on object

```
public void Udskriv()
{

 object[] msg={"This is year", 2004};
 Console.WriteLine("Message: {0} {1}", msg);
 Console.ReadLine();

}
```

Formatting flag

Example:

```
...

double price = etObj.VatCalc(25.35);

Console.WriteLine("Price incl. VAT: {0} ", price);
...
public double VatCalc(double inVal)
   {

         return inVal*1.25;  // Oh no…

   }
...
```

Result:

`Price incl. VAT`: 31,6875

- Can be formatted with
  - C for currency or
  - F2 (2 decimals)

C

```
Console.WriteLine("Price incl. VAT: {0:C} ", resultat);
```

`Price incl. VAT`: kr 31,69 (Notice: local currency)

F2

```
Console.WriteLine("Price incl. VAT: {0:F2} ", resultat);
```

`Price incl. VAT`: 31,69

Datatypes in C#

Table 3-4. *The Intrinsic Data Types of C#*

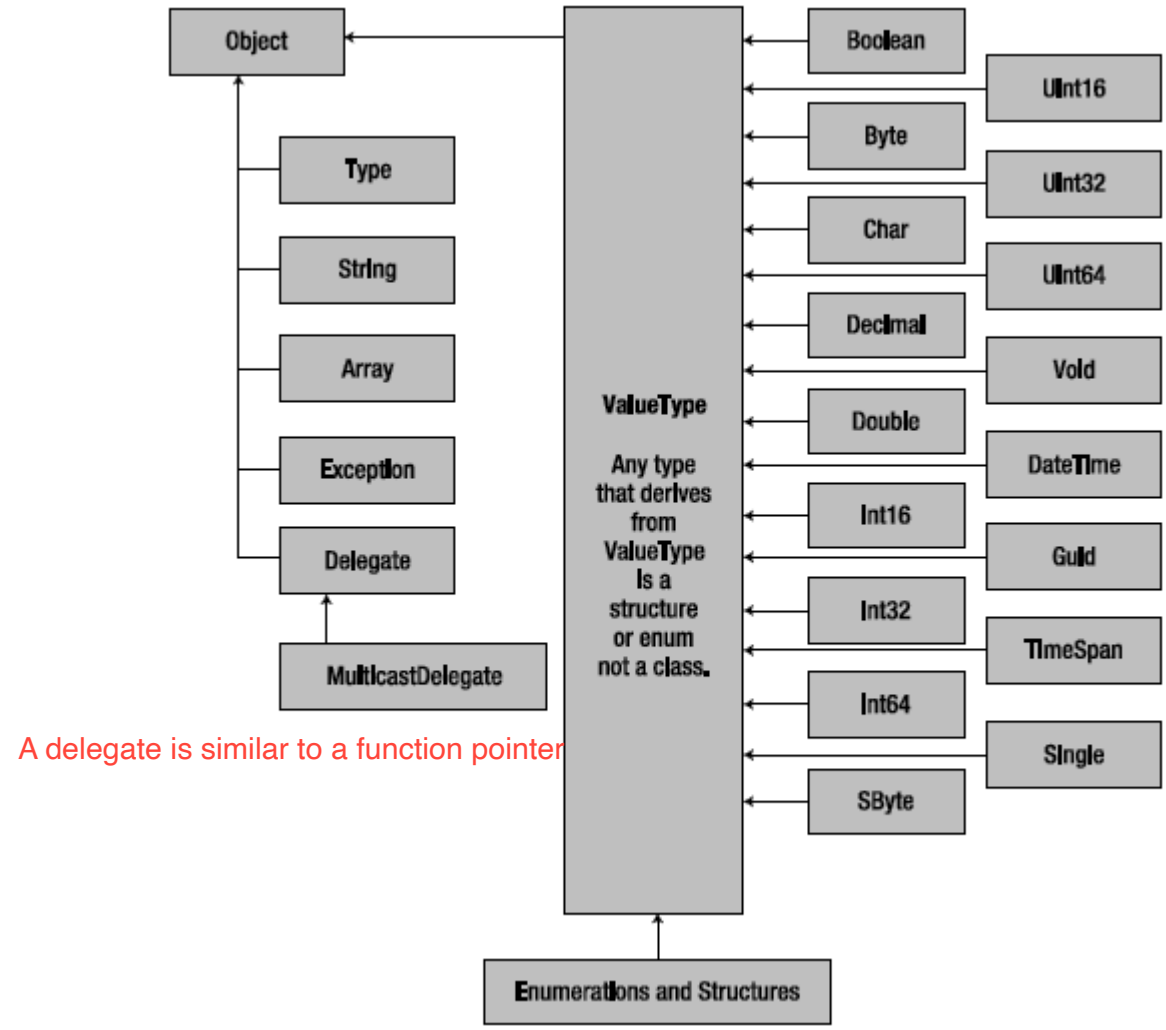
C# Shorthand	CLS Compliant?	System Type	Range	Meaning in Life
bool	Yes	System.Boolean	true or false	Represents truth or falsity
sbyte	No	System.SByte	–128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	–32,768 to 32,767	Signed 16-bit number
ushort	No	System.UInt16	0 to 65,535	Unsigned 16-bit number
int	Yes	System.Int32	–2,147,483,648 to 2,147,483,647	Signed 32-bit number
uint	No	System.UInt32	0 to 4,294,967,295	Unsigned 32-bit number
long	Yes	System.Int64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit number
ulong	No	System.UInt64	0 to 18,446,744,073,709,551, 615	Unsigned 64-bit number
char	Yes	System.Char	U+0000 to U+ffff	Single 16-bit Unicode character
float	Yes	System.Single	-3.4 5 10^{38} to +3.4 5 10^{38}	32-bit floating-point number
double	Yes	System.Double	±5.0 5 10^{-324} to ±1.7 5 10^{308}	64-bit floating-point number
decimal	Yes	System.Decimal	(-7.9 x 10^{28} to 7.9 x 10^{28}) / (10^{0} to ^{28})	128-bit signed number
string	Yes	System.String	Limited by system memory	Represents a set of Unicode characters
Object	Yes	System.Object	Can store any data type in an object variable	The base class of all types in the .NET universe

C# 9: New()

C# 9.0 adds a shortcut for creating variable instances. This shortcut is simply using the keyword new() without the data type. The updated version of NewingDataTypes is shown here:

```
static void NewingDataTypesWith9()
{
  Console.WriteLine("=> Using new to create variables:");
  bool b = new();              // Set to false.
  int i = new();                // Set to 0.
  double d = new();           // Set to 0.
  DateTime dt = new();     // Set to 1/1/0001 12:00:00 AM
  Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
  Console.WriteLine();
}
```

Datatype hierarchy



Value and reference types

- All native types (incl structs) is created on the stack (Value types)
- All class objects is created on the heap using new and uses reference

Example – reference:

```
using System;

public class RefPlay {

 private int x;

  public void setX(int inVal)

  {
```

```
      x = inVal;

  }

  public int getX()
  {

    return x;
  }

}

public class UseRefPlay {

  public static int Main()
  {

   RefPlay rf1 = new RefPlay();
   rf1.setX(22);

   RefPlay rf2 = rf1;
   rf2.setX(11);

   Console.WriteLine("res: {0} ", rf1.getX());
   Console.ReadLine();
   return 0;
  }

}
```

Result: res: 11

- Two references to the same object on the heap

System.Object class

- All data types (value or reference) inherits from System.Object
  - Some member functions are **virtual** and can be overrided

- Methods in System.Object:

- Equals()
- GetHashCode()
- GetType()
- ToString() (returns the name of the object)
- Finalize() (Is called when the object is removed from the heap)
- MemberWiseClone()

The keyword override

```
...
public override string ToString()
{
   StringBuilder sb = new StringBuilder();
   sb.AppendFormat("[FirstName= {0}", this.FirstName);
   sb. AppendFormat(" LastName= {0}", this.LastName);
   sb. AppendFormat(" SSN= {0}", this.SSN);
   sb. AppendFormat(" Age= {0}]", this.age);

   return sb.ToString();
}
...
```

Static members of System.Object

- Object.Equals() (Are two objects value equal?)
- Object.ReferenceEquals() (Has two objects the same address?)

```
...
Person p3 = new Person("Sally", "Jones","333", 4);
Person p4 = new Person("Sally", "Jones","333", 4);
Console.WriteLine("P3 and P4 have same state: {0}",   // OK
                          object.Equals(p3, p4));
Console.WriteLine("P3 and P4 are pointing to same object: {0}",
                          object.ReferenceEquals(p3, p4)); //No
...
```

More about Datatypes in C# (System)

All value types inheriting from ValueType are structs – not classes

- Notice: BigInteger
  - Is immutable (cannot be changed)
  - Can contain very-very big numbers
- Numerics namespace contains also a Complex struct

Max and min values

Example:

```
...

Console.WriteLine("max value {0}", UInt16.MaxValue);
...
```

Others:

- Double.Epsilon (smallest positive value greater than zero)
- Double.PositiveInfinity
- Double.NegativeInfinity

System.Boolean

- Can only have the values **true** and **false**
  - not for example : 1 and 0
- Uses bool.FalseString og bool.TrueString

System.Char

Methods (examples):

- char.IsDigit()
- char.IsLetter()
- char.IsWhiteSpace
- char.IsPunctuation()

Example char.IsWhiteSpace()

```
...

Console.WriteLine("whitespace test: {0}",
                  char.IsWhiteSpace("Hej der",3));
...
```

- returns True

Parsing data

- bool.Parse(“True”)
- double.Parse(“99.86”)
- int.Parse(“22”)
- char.Parse(“w”)

Example:

```
...
  int x = int.Parse("8");
  int y = int.Parse("12");
  int z = x+y;

  Console.WriteLine("resultat: {0} ", z);
...
```

TryParse (omitting try-catch)

```
static void ParseFromStringsWithTryParse()
{
  Console.WriteLine("=> Data type parsing with TryParse:");
  if (bool.TryParse("True", out bool b))
  {
    Console.WriteLine("Value of b: {0}", b);
  }
```

Digit separators (C# 7.2)

```
static void DigitSeparators()
{
  Console.WriteLine("=> Use Digit Separators:");
  Console.Write("Integer:");
  Console.WriteLine(123_456);
  Console.Write("Long:");
  Console.WriteLine(123_456_789L);
  Console.Write("Float:");
  Console.WriteLine(123_456.1234F);
  Console.Write("Double:");
  Console.WriteLine(123_456.12);
  Console.Write("Decimal:");
  Console.WriteLine(123_456.12M);
  //Updated in 7.2, Hex can begin with _
  Console.Write("Hex:");
  Console.WriteLine(0x_00_00_FF);
}
```

Binary separators

```
private static void BinaryLiterals()
{
  //Updated in 7.2, Binary can begin with _
  Console.WriteLine("=> Use Binary Literals:");
  Console.WriteLine("Sixteen: {0}",0b_0001_0000);
  Console.WriteLine("Thirty Two: {0}",0b_0010_0000);
  Console.WriteLine("Sixty Four: {0}",0b_0100_0000);
}
```

String class

- Most alike Java String class and (History: MFC CString class)
- A string cannot be changed (as in java)

Table 3-5. Select Members of System.String

String Member	Meaning in Life
Length	This property returns the length of the current string.
Compare()	This static method compares two strings.
Contains()	This method determines whether a string contains a specific substring.
Equals()	This method tests whether two string objects contain identical character data.
Format()	This static method formats a string using other primitives (e.g., numerical data, other strings) and the {0} notation examined earlier in this chapter.
Insert()	This method inserts a string within a given string.
PadLeft() PadRight()	These methods are used to pad a string with some characters.
Remove() Replace()	Use these methods to receive a copy of a string with modifications (characters removed or replaced).
Split()	This method returns a String array containing the substrings in this instance that are delimited by elements of a specified char array or string array.
Trim()	This method removes all occurrences of a set of specified characters from the beginning and end of the current string.
ToUpper() ToLower()	These methods create a copy of the current string in uppercase or lowercase format, respectively.

escape characters

- As in C++ and Java

Table 3-6. String Literal Escape Characters

Character	Meaning in Life
\'	Inserts a single quote into a string literal.
\"	Inserts a double quote into a string literal.
\\	Inserts a backslash into a string literal. This can be quite helpful when defining file or network paths.
\a	Triggers a system alert (beep). For console programs, this can be an audio clue to the user.
\n	Inserts a new line (on Windows platforms).
\r	Inserts a carriage return.
\t	Inserts a horizontal tab into the string literal.

String interpolation

```
static void StringInterpolation()
{
    // Some local variables we will plug into our larger string
    int age = 4;
    string name = "Soren";

    // Using curly-bracket syntax.
    string greeting = string.Format("Hello {0} you are {1} years old.", name, age);

    // Using string interpolation
    string greeting2 = $"Hello {name} you are {age} years old.";
}
```

Verbatim - @strings

Example:

```
...
     string verbatMan = @"C:\edu\ihk\cv\pcdn\verbat.cs";
     Console.WriteLine(verbatMan);
...
```

Output:

C:\edu\DTU\cv\pcdn\verbat.cs

```
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

Also note that verbatim strings can be used to preserve whitespace for strings that flow over multiple lines.

```
// Whitespace is preserved with verbatim strings.
string myLongString = @"This is a very
     very
          very
               long string";
Console.WriteLine(myLongString);
```

Using verbatim strings, you can also directly insert a double quote into a literal string by doubling the " token.

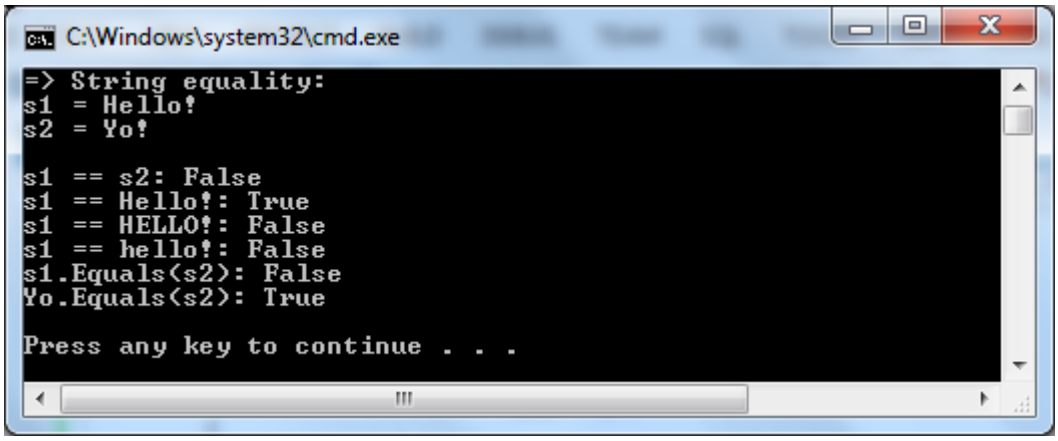
```
Console.WriteLine(@"Cerebus said ""Darrr! Pret-ty sun-sets""");
```

Verbatim strings can also be interpolated strings, by specifying both the interpolation operator ($) and the verbatim operator (@).

String comparison

```
static void StringEquality()
{
Console.WriteLine("=> String equality:");
string s1 = "Hello!";
string s2 = "Yo!";
Console.WriteLine("s1 = {0}", s1);
Console.WriteLine("s2 = {0}", s2);
Console.WriteLine();
// Test these strings for equality.
Console.WriteLine("s1 == s2: {0}", s1 == s2);
Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");
```

```
Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");
Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");
Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));
Console.WriteLine("Yo.Equals(s2): {0}", "Yo!".Equals(s2));
Console.WriteLine();
}
```



String Comparison enumeration

Table 3-7. *Values of the StringComparison Enumeration*

C# Equality/Relational Operator	Meaning in Life
`CurrentCulture`	Compares strings using culture-sensitive sort rules and the current culture
`CurrentCultureIgnoreCase`	Compares strings using culture-sensitive sort rules and the current culture and ignores the case of the strings being compared
`InvariantCulture`	Compares strings using culture-sensitive sort rules and the invariant culture
`InvariantCultureIgnoreCase`	Compares strings using culture-sensitive sort rules and the invariant culture and ignores the case of the strings being compared
`Ordinal`	Compares strings using ordinal (binary) sort rules
`OrdinalIgnoreCare`	Compares strings using ordinal (binary) sort rules and ignores the case of the strings being compared

System.Text.StringBuilder

- a string cannot be changed
- example. ToUpper returns a new string
  - NOT efficient (can result in many new strings)
  - StringBuilder can change a string
- In the namespace **System.Text**

- Use ToString() for conversion to a string

Datatype conversion and methods

- ”Widening”

```
short n1 = 19;
short n2 = 10;

Console.Writeline(”{0}”, Add(n1,n2) ) ; // Add takes two int’s as parameters and return an int

// Implicit widening
```

- ”Narrowing”

```
short n1 = 30000;
short n2 = 30000;
short n3 = Add(n1,n2);  // error
Console.Writeline(”{0}”, n3 ) ; // an int cannot be contained in  a short

// narrowing
```

Constants in C#

- uses the keyword **const**
  - is created on compile time
  - Cannot be used on references (are created on run-time)

- a class with constants (a container) can be created in two ways:
  - Make the constructor private
  - or the class abstract

Example:

```
using System;


class MineKonstanter {

 private MineKonstanter() { }

 public const int enConst = 2;

}

class UseConst {

 public static int Main() {
   // Does not work – construktor is private
  //MineKonstanter mk = new MineKonstanter();
  Console.WriteLine("En const {0}",
                    MineKonstanter.enConst);
                    // Use fully qualified name

  Console.ReadLine();

  return 0;
 }
```

or:

```
...

abstract class MineKonstanter {



 public const int enConst = 2;

}
...
```

General numeric type conversion

- It is of course possible to convert to a bigger type

```
...
byte b = 24;
int  x = b;
...
```

- otherwise is explicit type conversion used

```
...
int x = 257;
byte b = (byte) x;  // loss of information
...
```

Bad code example:

```
using System;

public class Ups
{

  public static void Main()

  {
         byte b1  = 200;
         byte b2  = 200;
         byte b3  = (byte) (b1+b2);
         Console.WriteLine("b3: {0}",b3);
         Console.ReadLine();
  }

}
```

Resul: b3: 144

Reason: 400 – 256 = 144

Solution: checked

```
using System;

public class Ups
{


  public static void Main()

  {
         byte b1  = 200;
         byte b2  = 200;

         try{

          byte b3  = checked( (byte) (b1+b2) );

          Console.WriteLine("b3: {0}",b3);
         }catch(OverflowException ofe)
         {

             Console.WriteLine(ofe.Message);
         }
         Console.ReadLine();
  }


}
```

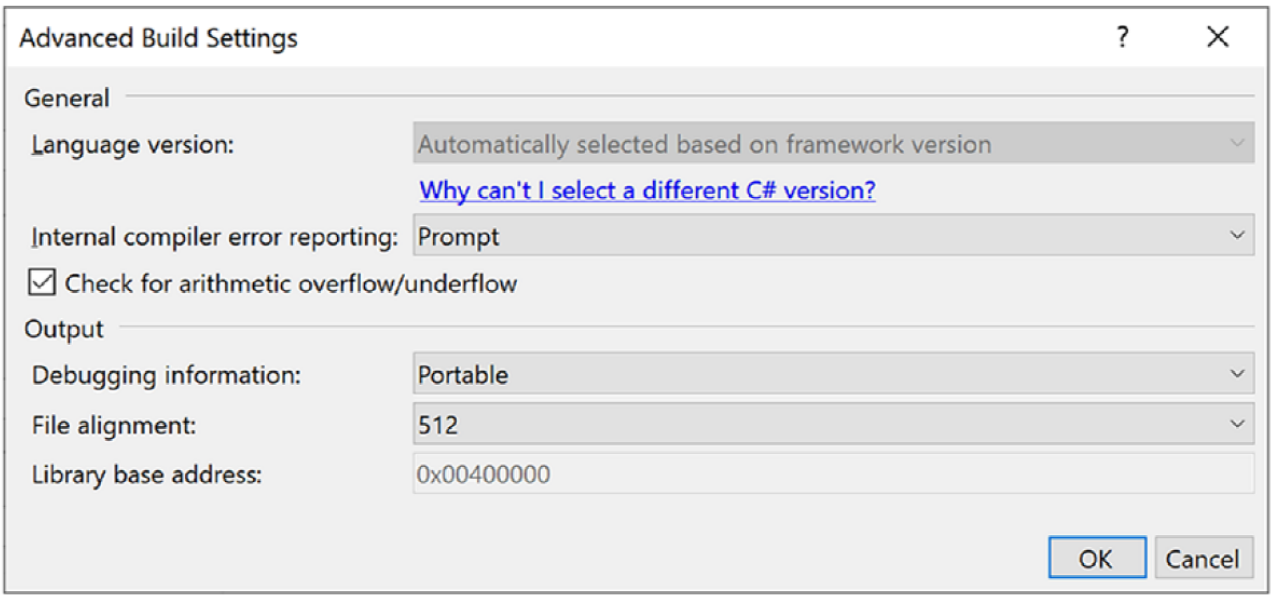
Output: Arithmetic operation resulted in an overflow.

- checked can be used on a block of code

```
...

  try{

           checked
           {
            b3   =   (byte) (b1+b2) ;
            Console.WriteLine("b3: {0}",b3);
            b4 =    (byte) (b1+b3) ;
            Console.WriteLine("b4: {0}",b4);
           }

          }catch(OverflowException ofe)
          {

              Console.WriteLine(ofe.Message);
          }
...
```

- Notice: BEGIN / END paranthesis

- General solution
  - Set Check in Project Settings



uncheck

- if checked is true in Project Settings
  - and allow some part to be unchecked

Example:

```
...

unchecked
{
         byte b1  = 200;
         byte b2  = 200;
         byte b3  = (byte) (b1+b2);
}
...
```

- Gives not a runtime exception
- No UnderflowException exists

```
...

public static void Main()

  {
         byte b1  = 0;
         byte b2  = 0;
         byte b3;


         try{

          checked
          {
           b3  =  (byte) (b1+b2-100) ;
           Console.WriteLine("b3: {0}",b3);
          }

         }catch(OverflowException ofe)
         {

             Console.WriteLine(ofe.Message);
         }

         Console.ReadLine();

  }
...
```

- Creates an OverflowException

var keyword (implicit typed variables)

```
...

var myInt = 0;
var myBool = true;
...
```

Example on **var**:

```
...

int[] tal= {1,2,3,4,5};

// LINQ query!

var delmængde = from i in tal where i > 3 select i;

// use foreach construction

...
```

Note Strictly speaking, `var` is not a C# keyword. It is permissible to declare variables, parameters, and fields named `var` without compile-time errors. However, when the `var` token is used as a data type, it is contextually treated as a keyword by the compiler.

Restrictions on var

```
private var tal = 22;     // cannot be used as field data
public var Metode(){}  // A method cannot return var
```

- Another example on var

```
static void LinqQueryOverInts()
{
int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
// LINQ query!
var subset = from i in numbers where i < 10 select i;
Console.Write("Values in subset: ");
foreach (var i in subset)
{
Console.Write("{0} ", i);
}
Console.WriteLine();
// Hmm...what type is subset?
Console.WriteLine("subset is a: {0}", subset.GetType().Name);
Console.WriteLine("subset is defined in: {0}", subset.GetType().Namespace);
}
```

Declaring Numerics Implicitly

As stated earlier, whole numbers default to integers, and floating-point numbers default to doubles. Create a new method named DeclareImplicitNumerics, and add the following code to demonstrate implicit declaration of numerics:

```
static void DeclareImplicitNumerics()
{
  // Implicitly typed numeric variables.
  var myUInt = 0u;
  var myInt = 0;
  var myLong = 0L;
  var myDouble = 0.5;
  var myFloat = 0.5F;
  var myDecimal = 0.5M;

  // Print out the underlying type.
  Console.WriteLine("myUInt is a: {0}", myUInt.GetType().Name);
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myLong is a: {0}", myLong.GetType().Name);
  Console.WriteLine("myDouble is a: {0}", myDouble.GetType().Name);
  Console.WriteLine("myFloat is a: {0}", myFloat.GetType().Name);
  Console.WriteLine("myDecimal is a: {0}", myDecimal.GetType().Name);
}
```

Iterations – foreach/ in

- as in java / C++
  - for construct
  - while
  - do-while
    - foreach in java is a specilized for construction

foreach example:

```
...

int[] mineTal={1,2,3,4,5,6,7,8,9,10};

  foreach( int res in mineTal)
  {
         if( res == 9)
            Console.WriteLine("Found 9");

  }
...
```

Relations and equality operators

Note C# 7 extends the `is` expression and `switch` statements with a technique called *pattern matching*. The basics of how these extensions affect `if/else` and `switch` statements are shown here for completeness. These extensions will make more sense after reading Chapter 6, which covers base class/derived class rules, casting, and the standard `is` operator.

- if operates only on boolean values (not as in C/C++)

THIS WILL NOT WORK:

```
...

string enStreng = ”C# is super”;
  if(enStreng.Length)
        // do something
...
```

This is OK:

```
...

string enStreng = ”C# is super”;
  if(0 != enStreng.Length)
        // do something
...
```

Table 3-8. *C# Relational and Equality Operators*

C# Equality/Relational Operator	Example Usage	Meaning in Life
==	if(age == 30)	Returns true only if each expression is the same
!=	if("Foo" != myStr)	Returns true only if each expression is different
<	if(bonus < 2000)	Returns true if expression A (bonus) is less than expression B (2000)
>	if(bonus > 2000)	Returns true if expression A (bonus) is greater than expression B (2000)
<=	if(bonus <= 2000)	Returns true if expression A (bonus) is less than or equal to expression B (2000)
>=	if(bonus >= 2000)	Returns true if expression A (bonus) is greater than or equal to expression B (2000)

Pattern matching if

```
static void IfElsePatternMatching()
{
  Console.WriteLine("===If Else Pattern Matching ===/n");
  object testItem1 = 123;
  object testItem2 = "Hello";
  if (testItem1 is string myStringValue1)
  {
    Console.WriteLine($"{myStringValue1} is a string");
  }
  if (testItem1 is int myValue1)
  {
    Console.WriteLine($"{myValue1} is an int");
  }
```

Ternary

The Conditional Operator (Updated 7.2)

The conditional operator (`?:`), also known as the Ternary Conditional Operator, is a shorthand method of writing a simple `if-else` statement. The syntax works like this:

```
condition ? first_expression : second_expression;
```

The condition is the conditional test (the `if` part of the `if-else` statement). If the test passes, then the code immediately after the question mark (`?`) is executed. If the test does not evaluate to true, the code after the colon (the `else` part of the `if-else` statement) is executed. The previous code example can be written using the conditional operator like this:

```
private static void ExecuteIfElseUsingConditionalOperator()
{
  string stringData = "My textual data";
  Console.WriteLine(stringData.Length > 0
    ? "string is greater than 0 characters"
    : "string is not greater than 0 characters");
  Console.WriteLine();
}
```

Logical Operators

An `if` statement may be composed of complex expressions as well and can contain `else` statements to perform more complex testing. The syntax is identical to C (and C++) and Java. To build complex expressions, C# offers an expected set of logical operators, as shown in Table 3-9.

Table 3-9. *C# Logical Operators*

Operator	Example	Meaning in Life
&&	`if(age == 30 && name == "Fred")`	AND operator. Returns `true` if all expressions are true
||	`if(age == 30 || name == "Fred")`	OR operator. Returns `true` if at least one expression is true
!	`if(!myBool)`	NOT operator. Returns `true` if false, or `false` if true

Note The && and || operators both "short-circuit" when necessary. This means that after a complex expression has been determined to be false, the remaining subexpressions will not be checked. If you require all expressions to be tested regardless, you can use the related & and | operators.

Switch

```
    switch (n)
    {
      case 1:
        Console.WriteLine("Good choice, C# is a fine language.");
        break;
      case 2:
        Console.WriteLine("VB: OOP, multithreading, and more!");
        break;
      default:
        Console.WriteLine("Well...good luck with that!");
        break;
    }
}
```

Note C# demands that each case (including `default`) that contains executable statements have a terminating `return`, `break`, or `goto` to avoid falling through to the next statement.

Strings in switch

---

a string variable:

```
static void SwitchOnStringExample()
{
  Console.WriteLine("C# or VB");
  Console.Write("Please pick your language preference: ");

  string langChoice = Console.ReadLine();
  switch (langChoice.ToUpper())
  {
    case "C#":
      Console.WriteLine("Good choice, C# is a fine language.");
      break;
    case "VB":
      Console.WriteLine("VB: OOP, multithreading and more!");
      break;
    default:
      Console.WriteLine("Well...good luck with that!");
      break;
  }
```

Pattern matching

Switch Statement Pattern Matching (New 7.0)

Prior to C# 7, match expressions in `switch` statements were limited to comparing a variable to constant values, sometimes referred to as the *constant pattern*. In C# 7, `switch` statements can also employ the *type pattern*, where `case` statements can evaluate the *type* of the variable being checked and `case` expressions are no longer limited to constant values. The rule that each `case` statement must be terminated with a `return` or `break` still applies; however, `goto` statements are not supported using the type pattern.

Note If you are new to object-oriented programming, this section might be a little confusing. It will all come together in Chapter 6, when you revisit the new pattern matching features of C# 7 in the context of classes and base classes. For now, just understand that there is a powerful new way to write `switch` statements.

```
//This is new the pattern matching switch statement
switch (choice)
{
  case int i:
    Console.WriteLine("Your choice is an integer.");
    break;
  case string s:
    Console.WriteLine("Your choice is a string.");
    break;
  case decimal d:
    Console.WriteLine("Your choice is a decimal.");
    break;
  default:
    Console.WriteLine("Your choice is something else");
    break;
}
```

```
  switch (choice)
  {
    case int i when i == 2:
    case string s when s.Equals("VB", StringComparison.OrdinalIgnoreCase):
      Console.WriteLine("VB: OOP, multithreading, and more!");
      break;
    case int i when i == 1:
    case string s when s.Equals("C#", StringComparison.OrdinalIgnoreCase):
      Console.WriteLine("Good choice, C# is a fine language.");
      break;
    default:
      Console.WriteLine("Well...good luck with that!");
      break;
  }
  Console.WriteLine();
}
```

This adds a new dimension to the `switch` statement as the order of the `case` statements is now significant. With the constant pattern, each `case` statement had to be unique. With the type pattern, this is no longer the case. For example, the following code will match every integer in the first case statement and will never execute the second or the third (in fact, the following code will fail to compile):

C#8

With the new switch expressions in C# 8, the previous method can be written as follows, which is much more concise:

```
static string FromRainbow(string colorBand)
{
  return colorBand switch
      "Red" => "#FF0000",
      "Orange" => "#FF7F00",
      "Yellow" => "#FFFF00",
      "Green" => "#00FF00",
      "Blue" => "#0000FF",
      "Indigo" => "#4B0082",
      "Violet" => "#9400D3",
      _ => "#FFFFFF",
    };
  }
```

There is a lot to unpack in that example, from the lambda (=>) statements to the discard (_). These will all be covered in later chapters, as will this example in further detail.

C# 9: Pattern matching

Making Pattern Matching Improvements (New 9.0)

C# 9.0 has introduced a host of improvements to pattern matching, as shown in Table 3-9.

Table 3-9. *Pattern Matching Improvements*

Pattern	Meaning in Life
`Type patterns`	Checks if a variable is a type
`Parenthesized patterns`	Enforces or emphasizes the precedence of pattern combinations
`Conjuctive (and) patterns`	Requires both patterns to match
`Disjunctive (or) patterns`	Requires either pattern to match
`Negated (not) patterns`	Requires a pattern does not match
`Relational patterns`	Requires input to be less than, less than or equal, greater than, or greater than or equal

Examples from the book:

```
static void IfElsePatternMatchingUpdatedInCSharp9()
{
    Console.WriteLine("================ C# 9 If Else Pattern Matching Improvements
==============/n");
    object testItem1 = 123;
    Type t = typeof(string);
    char c = 'f';

    //Type patterns
    if (t is Type)
    {
        Console.WriteLine($"{t} is a Type");
    }

    //Relational, Conjuctive, and Disjunctive patterns
    if (c is >= 'a' and <= 'z' or >= 'A' and <= 'Z')
    {
        Console.WriteLine($"{c} is a character");
    };
```

```
    //Parenthesized patterns
    if (c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',')
    {
        Console.WriteLine($"{c} is a character or separator");
    };

    //Negative patterns
    if (testItem1 is not string)
    {
        Console.WriteLine($"{testItem1} is not a string");
    }
    if (testItem1 is not null)
    {
        Console.WriteLine($"{testItem1} is not null");
    }
    Console.WriteLine();
}
```

Using tuples

```
//Switch expression with Tuples
static string RockPaperScissors(string first, string second)
{
  return (first, second) switch
  {
    ("rock", "paper") => "Paper wins.",
    ("rock", "scissors") => "Rock wins.",
    ("paper", "rock") => "Paper wins.",
    ("paper", "scissors") => "Scissors wins.",
    ("scissors", "rock") => "Rock wins.",
    ("scissors", "paper") => "Scissors wins.",
    (_, _) => "Tie.",
  };
}
```

Access modifiers – methods

Modifier	
Public	Object level
Private	(default) Internal in the class
protected	Private in an inheritance hierarchy
internal	Access in the assembly –not outside
protected internal	Defines a method with access limited to current assembly or types derived from a defining class in the current assembly (OR'ing)

- private
  - Constructors Can be used to create a singleton
  - Helper functions

Static methods

- A static method exists on class level, example:
  - Main
  - WriteLine()

Static classes

- Static classes can only contain static methods and data

Static data

- non-static data
  - every instance has a copy
- static data
  - Common data example:
    - Object counters

Modification of parameters in methods

- Analog to IDL (Interface Definition Language)
- Parameters are normally transferred ”by value”

- out
  - Can receive multiple return values in a call to a method
  - Should not be initialized before call to method
- ref
  - Value placed by the caller.
  - Value can be changed by the method
  - MUST BE initialized before calling
- params
  - params gives a variable number of arguments
  - Flexible!!

Example on params:

```
...
public static void udskriv(string info,params int[] list) {

     Console.WriteLine(info + "\n");

     for( int i = 0; i < list.Length;i++)
        Console.WriteLine(list[i]);

}
...
// Use of methods
  udskriv("Data:",1,2,3,4);
  udskriv("Other Data:", 0,44);

...
```

”By reference” and ”By value”

- ref keyword
  - ref **allows** manipulation of the parameter

Example:

```
using System;



class Elektronik {

public string compType;  // public on purpose
public double pris;      // public on purpose


public Elektronik(string name, double p)
{
        compType = name;

        pris     = p;
1

}


class Lager {


     public static int Main() {

     Elektronik c1 = new Elektronik("IC1",120.25);
     ListComp( ref c1);
     Elektronik c2 = new Elektronik("IC2",8.25);
     ListComp2(c2);
     udskriv(c1);
     udskriv(c2);


     Console.ReadLine();

     return 0;
    }


    public static void ListComp( ref Elektronik e) {

      e.compType="BLA1";
      e = new Elektronik("BLA2",0.0);
     // A new object on the heap
     // e IS c1
```

```
    }

   public static void ListComp2( Elektronik e) {

      e.compType="BLA3";
      e = new Elektronik("BLA4",0.0);
      //Will be forgotten after the call to the method
      // e IS A COPY OF c2



    }

    public static void udskriv(Elektronik e) {

     Console.WriteLine("Name: {0},
                    prize: {1}",e.compType,e.pris);

    }

}



}
```

Output:

Name: BLA2, prize: 0
Name: BLA3, prize: 8,25

- When using **ref:**
  - The method can change object data
  - And change object reference (refer to another object)

Optional parameters

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Opt("Hi");
            Opt("Hallo","A Provider");
            Console.ReadLine();
        }

        static void Opt(String msg, string provider = "SomeProvider")
        {
            Console.WriteLine(msg + "\n" + provider);
        }
    }
}
```

Named parameters (from 4.0 platform)

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Opt("Hi");
            Opt("Hallo","A Provider");
            Named(msg: "Named param",antal:44);  // use a named parameter

            Console.ReadLine();
        }

        static void Opt(String msg, string provider = "SomeProvider")
        {
            Console.WriteLine(msg + "\n" + provider);
        }

        static void Named(String msg, int antal)
        {
```

```
            Console.WriteLine(msg);
            Console.WriteLine(antal);
        }
    }
}
```

Method overload

- Same method name - different parameter list

```
class Program
{
static void Main(string[] args)
{
}
// Overloaded Add() method.
static int Add(int x, int y)
{ return x + y; }
static double Add(double x, double y)
{ return x + y; }
static long Add(long x, long y)
{ return x + y; }
}
```

Using Lambda expressions:

Expression-Bodied Members

You already learned about simple methods that return values, such as the `Add()` method. C# 6 introduced expression-bodied members that shorten the syntax for single-line methods. For example, `Add()` can be rewritten using the following syntax:

```
static int Add(int x, int y) => x + y;
```

This is what is commonly referred to as *syntactic sugar*, meaning that the generated IL is no different. It's just another way to write the method. Some find it easier to read, and others don't, so the choice is yours (or your team's) which style you prefer.

Note Don't be alarmed by the `=>` operator. This is a lambda operation, which is covered in great detail in Chapter 12. That chapter also explains exactly *how* expression-bodied members work. For now, just consider them a shortcut to writing single-line statements.

Local functions

```
static int AddWrapper(int x, int y)
{
  //Do some validation here
  return Add();

  int Add()
  {
    return x + y;
  }
}
```

Static local functions

Static Local Functions (New 8.0)

An improvement to local functions that was introduced in C# 8 is the ability to declare a local function as static. In the previous example, the local Add() function was referencing the variables from the main function directly. This could cause unexpected side effects, since the local function can change the values of the variables.

To see this in action, create a new method call AddWrapperWithSideEffect(), as shown here:

```
static int AddWrapperWithSideEffect(int x, int y)
{
  //Do some validation here
  return Add();

  int Add()
  {
    x += 1;
    return x + y;
  }
}
```

Solution:

The improved version of the previous method is shown here:

```
static int AddWrapperWithStatic(int x, int y)
{
  //Do some validation here
  return Add(x,y);

  static int Add(int x, int y)
  {
    return x + y;
  }
}
```

Out modifier update in C# 7:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
...
  // No need to assign initial value to local variables
  // used as output parameters, provided the first time
  // you use them is as output arguments.
  // C# 7 allows for out parameters to be declared in the method call
  Add(90, 90, out int ans);
  Console.WriteLine("90 + 90 = {0}", ans);
  Console.ReadLine();
}
```

Discards

Discards (New 7.0)

If you don't care about the value of an `out` parameter, you can use a discard as a placeholder. Discards are temporary, dummy variables that are intentionally unused. They are unassigned, don't have a value, and might not even allocate any memory. This can provide a performance benefit as well as make your code more readable. Discards can be used with `out` parameters, with tuples (later in this chapter), with pattern matching (Chapters 6 and 8), or even as stand-alone variables.

For example, if you want to get the value for the int in the previous example but don't care about the second two parameters, you could write the following code:

```
//This only gets the value for a, and ignores the other two parameters
FillTheseValues(out int a, out _, out _);
```

Note that the called method is still doing the work setting the values for all three parameters; it's just that the last two parameters are being *discarded* when the method call returns.

The in modifier
(problem here)

The in Modifier (New 7.2)

The in modifier passes a value by reference (for both value and reference types) and prevents the called method from modifying the values. This clearly states a design intent in your code, as well as potentially reducing memory pressure. When value types are passed by value, they are copied (internally) by the called method. If the object is large (such as a large struct), the extra overhead of making a copy for local use can be significant. Also, even when reference types are passed without a modifier, they can be modified by the called method. Both of these issues can be resolved using the in modifier.

Revisiting the Add method from earlier, there are two lines of code that modify the parameters, but don't effect the values for the calling method. The values aren't effected because the Add method makes a copy of the variables x and y to use locally. While the calling method doesn't have any adverse side effects, what if the Add method was changed to this code:

```
static int Add2(int x,int y)
{
  x = 10000;
  y = 88888;
  int ans = x + y;
  return ans;
}
```

Running this code then returns 98888, regardless of the numbers sent into the method. This is obviously

Solution:

```
static int AddReadOnly(in int x,in int y)
{
  //Error CS8331 Cannot assign to variable 'in int' because it is a readonly variable
  //x = 10000;
  //y = 88888;
```

Array manipulation

- Very much alike C++ and Java
  - Using zero index

Example:

```
...

string[] str = new string[10];
...
```

- Notice the use of new

This won't work:

```
...

// OOPS! Mismatch of size and elements!
int[] intArray = new int[2] { 20, 22, 23, 0 };

...
```

This is OK:

```
...

int[] intArray = new int[4] { 20, 22, 23, 0 };
...
```

- Compiler decides the size of the array

Implicitly Typed Local Arrays

In Chapter 3, you learned about the topic of implicitly typed local variables. Recall that the var keyword allows you to define a variable, whose underlying type is determined by the compiler. In a similar vein, the var keyword can be used to define *implicitly typed local arrays*. Using this technique, you can allocate a new array variable without specifying the type contained within the array itself (note you must use the new keyword when using this approach).

```
static void DeclareImplicitArrays()
{
  Console.WriteLine("=> Implicit Array Initialization.");

  // a is really int[].
  var a = new[] { 1, 10, 100, 1000 };
  Console.WriteLine("a is a: {0}", a.ToString());

  // b is really double[].
  var b = new[] { 1, 1.5, 2, 2.5 };
```

Arrays of objects

```
static void ArrayOfObjects()
{
  Console.WriteLine("=> Array of Objects.");

  // An array of objects can be anything at all.
  object[] myObjects = new object[4];
  myObjects[0] = 10;
  myObjects[1] = false;
  myObjects[2] = new DateTime(1969, 3, 24);
  myObjects[3] = "Form & Void";
  foreach (object obj in myObjects)
  {
    // Print the type and value for each item in array.
    Console.WriteLine("Type: {0}, Value: {1}", obj.GetType(), obj);
  }
```

Methods and arrays

- A method can (of course) have an array as parameter

```
...
public static void EnMetode( int[] nogleInts)
{

}
...
```

- And can return an array

```
...
public static int[] EnMetode( int[] nogleInts)
{

  return nogleInts;
}
...
```

Multi dimensionel arrays

- As in C++ and java

Two types of multi dimensional arrays:

- Array of arrays
- Rectangular arrays

Example (rectangular arrays):

```
...

int[,] myMatrix;
myMatrix = new int[6,6];

// Populate (6 * 6) array.
for(int i = 0; i < 6; i++)
  for(int j = 0; j < 6; j++)
     myMatrix[i, j] = i * j;

          // Show (6 * 6) array.
          for(int i = 0; i < 6; i++)
          {
             for(int j = 0; j < 6; j++)
             {

                Console.Write(myMatrix[i, j] + "\t");
             }
            Console.WriteLine();

          }
...
```

- All arrays inherit from System.Array

Methods on an array – system.Array

***Table** 4-2. Select Members of System.Array*

Member of Array Class	Meaning in Life
Clear()	This static method sets a range of elements in the array to empty values (0 for numbers, null for object references, false for booleans).
CopyTo()	This method is used to copy elements from the source array into the destination array.
Length	This property returns the number of items within the array.
Rank	This property returns the number of dimensions of the current array.
Reverse()	This static method reverses the contents of a one-dimensional array.
Sort()	This static method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the IComparer interface, you can also sort your custom types (see Chapter 9).

Indices and Ranges (New 8.0)

To simplify working with sequences (including arrays), C# 8 introduces two new types and two new operators for use when working with arrays:

- System.Index represents an index into a sequence.
- System.Range represents a subrange of indices.
- The index from end operator ^ specifies that the index is relative to the end of the sequence.
- The range operator (...) specifies the start and end of a range as its operands.

Note Indices and ranges can be used with arrays, strings, Span<T>, and ReadOnlySpan<T>.

Example

As you have already seen, arrays are indexed beginning with zero (0). The end of a sequence is the length of the sequence – 1. The previous for loop that printed the gothicBands array can be updated to the following:

```
for (int i = 0; i < gothicBands.Length; i++)
{
  Index idx = i;
  // Print a name
  Console.Write(gothicBands[idx] + ", ");
```

End operator

The index from end operator lets you specify how many positions from the end of sequence, starting with the length. Remember that the last item in a sequence is one less than the actual length, so ^0 would actually cause an error. The following code prints the array in reverse:

```
for (int i = 1; i <= gothicBands.Length; i++)
{
  Index idx = ^i;
  // Print a name
  Console.Write(gothicBands[idx] + ", ");
}
```

Ranges

```
foreach (var itm in gothicBands[0..2])
{
  // Print a name
  Console.Write(itm + ", ");
}
Console.WriteLine("\n");
```

Ranges can also be passed to a sequence using the new Range data type, as shown here:

```
Range r = 0..2; //the end of the range is exclusive
foreach (var itm in gothicBands[r])
{
  // Print a name
  Console.Write(itm + ", ");
}
Console.WriteLine("\n");
```

Enumerations

- symbolic names for numerical values
- Elements do not have to be sequential

Example:

```
...

  enum ProcessorType :byte
     {
       Pic12 = 12,
       Pic16 = 16
     }
...

Console.WriteLine("{0} ={1}",ProcessorType.Pic16, (int)
                             ProcessorType.Pic16);
...
```

Output: Pic16 =16

- Restricted to bytes in the example
- Can of course be used in a switch construction

```
static void AskForBonus(EmpType e)
{
switch (e)
{
case EmpType.Manager:
Console.WriteLine("How about stock options instead?");
break;
case EmpType.Grunt:
Console.WriteLine("You have got to be kidding...");
break;
case EmpType.Contractor:
Console.WriteLine("You already get enough cash...");
break;
case EmpType.VicePresident:
Console.WriteLine("VERY GOOD, Sir!");
break;
}
}
```

Finding an enum type

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

namespace ConsoleApplication1
{

    class Program
    {
        enum ProcessorType : byte { Pic12 = 12, Pic16 = 16 }

        static void Main(string[] args)
        {
            ProcessorType pt = ProcessorType.Pic16;
```

```
            Console.WriteLine(Enum.GetUnderlyingType(pt.GetType()));

            Console.ReadLine();
        }

    }
}
```

Dynamically Discovering an enum's Name-Value Pairs

Beyond the `Enum.GetUnderlyingType()` method, all C# enumerations support a method named `ToString()`, which returns the string name of the current enumeration's value. The following code is an example:

```
static void Main(string[] args)
{
  EmpTypeEnum emp = EmpTypeEnum.Contractor;
  ...
  // Prints out "emp is a Contractor".
  Console.WriteLine("emp is a {0}.", emp.ToString());
  Console.ReadLine();
}
```

GetValues()

System.Enum also defines another static method named GetValues(). This method returns an instan of System.Array. Each item in the array corresponds to a member of the specified enumeration. Consider the following method, which will print out each name-value pair within any enumeration you pass in as a parameter:

```
// This method will print out the details of any enum.
static void EvaluateEnum(System.Enum e)
{
  Console.WriteLine("=> Information about {0}", e.GetType().Name);

  Console.WriteLine("Underlying storage type: {0}",
    Enum.GetUnderlyingType(e.GetType()));

  // Get all name-value pairs for incoming parameter.
  Array enumData = Enum.GetValues(e.GetType());
  Console.WriteLine("This enum has {0} members.", enumData.Length);

  // Now show the string name and associated value, using the D format
  // flag (see Chapter 3).
  for(int i = 0; i < enumData.Length; i++)
```

Structs –"Light weight" classes in C#

- Has class behavior
- Can have constructors (ONLY with arguments)
- Can implement interfaces

- Is created on the stack
  - Even if you use new keyword
  - Stack PERFORMANCE IS BETTER

Example:

```
using System;


struct Komponent {

public string type;
public string navn;

public Komponent(string t, string n)
{
     type = t;
     navn = n;

}

}

class StructTest {


public static int Main()
{
  Komponent komp = new Komponent("IC","PIC16F876");
  udskriv(komp);

  return 0;
}

public static void udskriv(Komponent k)
{

    Console.WriteLine("NAVN: {0} TYPE: {1}",
                           k.navn,k.type);

}

}
```

Readonly Structs (New 7.2)

Structs can also be marked as readonly if there is a need for them to be *immutable*. Immutable objects must be set up at construction and, because they cannot be changed, can be more performant. When declaring a struct as readonly, all of the properties must also be readonly. But, you might ask, how can a property be set (as all properties must be on a struct) if it's read-only? The answer is that the value must be set during the construction of the struct.

Update the point class to the following example:

```
readonly struct ReadOnlyPoint
{
  // Fields of the structure.
  public int X {get; }
  public int Y { get; }

  // Display the current position and name.
  public void Display()
  {
    Console.WriteLine($"X = {X}, Y = {Y}");
  }


   public ReadOnlyPoint(int xPos, int yPos, string name)
   {
     X = xPos;
     Y = yPos;
   }
}
```

Readonly members (New 8.0)

New in C# 8.0, you can declare individual fields of a struct as `readonly`. This is more granular than making the entire struct readonly. The readonly modifier can be applied to methods, properties, and property accessors. Add the following struct code to your file, outside of the Program class:

```
struct PointWithReadOnly
{
  // Fields of the structure.
  public int X;
  public readonly int Y;
  public readonly string Name;
```

ref Structs (New 7.2)

Also added in C# 7.2, the ref modifier can be used when defining a struct. This requires all instances of the struct to be stack allocated and cannot be assigned as a property of another class. The technical reason for this is that ref structs cannot referenced from the heap. The difference between the stack and the heap is covered in the next section.

There are some additional limitations of ref structs:

- They cannot be assigned to a variable of type object, dynamic, or an interface type.
- They cannot implement interfaces.
- They cannot be used as a property of a non-ref struct.
- They cannot be used in async methods, in iterators, lambda expressions, or local functions.

Disposable ref Structs (New 8.0)

As covered in the previous section, ref structs (and readonly ref structs) cannot implement an interface and therefore cannot implement IDisposable. New in C# 8.0, ref structs and readonly ref structs can be made disposable by adding a public void Dispose() method.

Add the following struct definition to the main file:

```
ref struct DisposableRefStruct
{
  public int X;
  public readonly int Y;
  public readonly void Display()
  {
    Console.WriteLine($"X = {X}, Y = {Y}");
  }
  // A custom constructor.
  public DisposableRefStruct(int xPos, int yPos)
  {
    X = xPos;
    Y = yPos;
    Console.WriteLine("Created!");
  }
  public void Dispose()
  {
    //clean up any resources here
```

Converting between value – and reference types

Boxing:

- explicit converting a value type to a reference type
- Allocates a new object on the heap

Unboxing:

- Converting a value in a reference type to a value type on the stack

Example:

```
...
   int enInt = 22;
   object objInt = enInt;          // boxing
   int enAndenInt = (int) objInt; // unboxing
   Console.WriteLine("Here is enAndenInt {0}",
                        enAndenInt);
...
```

How often do you need the boxing – unboxing?

- Very rare
- The compiler performs boxing automatically when necessary

C# Nullable types (C# 7)

- Value types cannot have the value *null*
    - bool bTest = null; // ERROR!

- But for example Strings are reference types
    - string s = null; // OK

Since .net 2.0 it is possible to create nullable types!

- Useful in connection with databases
- Nullable types can contain normal value + **null**

Nullable types is created with ?

Examples:

```
int? nullInt = 10;
char? nullChar = ’a’;
```

Local nullable types must still have a value!

```
static void LocalNullableVariables()
{
// Define some local nullable variables.
int? nullableInt = 10;
double? nullableDouble = 3.14;
bool? nullableBool = null;
char? nullableChar = 'a';
int?[] arrayOfNullableInts = new int?[10];
// Error! Strings are reference types!
// string? s = "oops";
}
```

- Nullable types of the System.Nullable<T> struct type

```
static void LocalNullableVariablesUsingNullable()
{
// Define some local nullable types using Nullable<T>.
Nullable<int> nullableInt = 10;
Nullable<double> nullableDouble = 3.14;
Nullable<bool> nullableBool = null;
Nullable<char> nullableChar = 'a';
Nullable<int>[] arrayOfNullableInts = new int?[10];
}
```

?? operator

- Apply a value if a method returns null

```
...
int data = dr.GetIntFromDatabase() ?? 100;
...
```

C# Nullable REFERENCE types (C# 8)

Opt in

Nullable Reference Types (New 8.0)

A significant change in C# 8 is the support for nullable reference types. In fact, the change is so significant that the .NET Framework could not be updated to support this new feature. Hence, the decisions to only support C# 8 in .NET Core 3.0 and later and the decision that support for nullable reference types is an opt in. By default, when you create a new project in .NET Core 3.0/3.1, reference types work the same way that they did on C# 7. This is in order to prevent breaking billions of lines of code that exist in the pre-C# 8 ecosystem.

Opting in for Nullable Reference Types

Support for nullable reference types is controlled by setting a Nullable Context. This can be as big as an entire project (by updating the project file) or as small as a few lines (by using compiler directives). There are also two contexts that can be set:

- Nullable Annotation Context: This enables/disables the nullable annotation (?) for nullable reference types.
- Nullable Warning Context: This enables/disables the compiler warnings for nullable reference types.

Edit project file

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Table 4-4. *Values for Nullable in Project Files*

Value	Meaning in Life
Enable	Nullable Annotations are enabled and Nullable Warnings are enabled.
Warnings	Nullable Annotations are disabled and Nullable Warnings are enabled.
Annotations	Nullable Annotations are enabled and Nullable Warnings are disabled.
Disable	Nullable Annotations are disabled and Nullable Warnings are disabled.

Action:

```
public class TestClass
{
  public string Name { get; set; }
  public int Age { get; set; }
}
```

As you can see, this is just a normal class. The nullability comes in when you use this class in your code. Take the following declarations:

```
string? nullableString = null;
TestClass? myNullableClass = null;
```

The project file setting makes the entire project a nullable context. The nullable context allows the declarations of the string and TestClass types to use the nullable annotation (?). The following line of code generates a warning (CS8600) due to the assignment of a null to a non-nullable type in a nullable context:

```
//Warning CS8600 Converting null literal or possible null value to non-nullable type
TestClass myNonNullableClass = myNullableClass;
```

(must use the notation)

Fine tuning:

For finer control of where the nullable contexts are in your project, you can use compiler directives (as discussed earlier) to enable or disable the context. The following code turns off the nullable context (set at the project level) and then re-enables it by restoring the project settings:

```
#nullable disable
TestClass anotherNullableClass = null;
//Warning CS8632 The annotation for nullable reference types should only be used in code
within a '#nullable' annotations
TestClass? badDefinition = null;
```

The Null-Coalescing Assignment Operator (New 8.0)

Building on the null-coalescing operator, C# 8 introduced the *null-coalescing assignment operator* (??=). This operator assigns the left-hand side to the right-hand side only if the left-hand side is null. For example, enter the following code:

```
//Null-coalescing assignment operator
int? nullableInt = null;
nullableInt ??= 12;
nullableInt ??= 14;
Console.WriteLine(nullableInt);
```

The nullableInt variable is initialized to null. The next line assigns the value of 12 to the variable since the left-hand side is indeed null. The next line does *not* assign 14 to the variable, since it isn't null.

The Null conditional operator

```
Console.WriteLine($"You sent me {args?.Length ?? 0} arguments.");
```

Tuples

Tuples (New/Updated 7.0)

To wrap up this chapter, let's examine the role of tuples using a Console Application project named FunWithTuples. As mentioned earlier in this chapter, one way to use `out` parameters is to retrieve more than one value from a method call. Another way is to use a very light construct call a tuple.

Tuples are lightweight data structures that contain multiple fields. They were actually added to the language in C# 6, but in a very limited way. There was also a potentially significant problem with the C# 6 implementation: each field is implemented as a reference type, potentially creating memory and/or performance problems (from boxing/unboxing).

In C# 7, tuples use the new `ValueTuple` data type instead of reference types, potentially saving significant memory. The `ValueTuple` data type creates different structs based on the number of properties for a tuple. An additional feature added in C# 7 is that each property in a tuple can be assigned a specific name (just like variables), greatly enhancing the usability.

There are two important considerations for tuples:

- The fields are not validated.
- You cannot define your own methods.

They are really designed to just be a lightweight data transport mechanism.

Example

```
("a", 5, "c")
```

Notice that they don't all have to be the same data type. The parenthetical construct is also used to assign the tuple to a variable (or you can use the `var` keyword and the compiler will assign the data types for you). To assign the previous example to a variable, the following two lines achieve the same thing. The `values` variable will be a tuple with two `string` properties and an `int` property sandwiched in between.

```
(string, int, string) values = ("a", 5, "c");
var values = ("a", 5, "c");
```

By default, the compiler assigns each property the name `ItemX`, where `X` represents the one-based position in the tuple. For the previous example, the property names are `Item1`, `Item2`, and `Item3`. Accessing them is done as follows:

```
Console.WriteLine($"First item: {values.Item1}");
Console.WriteLine($"Second item: {values.Item2}");
Console.WriteLine($"Third item: {values.Item3}");
```

Inferred Variable Names (Updated 7.1)

An update to tuples in C# 7.1 is the ability for C# to infer the variable names of tuples, as is shown here:

```
Console.WriteLine("=> Inferred Tuple Names");
var foo = new {Prop1 = "first", Prop2 = "second"};
var bar = (foo.Prop1, foo.Prop2);
Console.WriteLine($"{bar.Prop1};{bar.Prop2}");
```

Tuple Equality/Inequality (New 7.3)

An added feature in C# 7.1 is the tuple equality (==) and inequality (!=). When testing for inequality, the comparison operators will perform implicit conversions on datatypes within the tuples, including comparing nullable and non-nullable tuples and/or properties. That means the following tests work perfectly, despite the difference between int/long:

Return values from methods:

By using a tuple, you can remove the parameters and still get the three values back.

```
static (int a,string b,bool c) FillTheseValues()
{
  return (9,"Enjoy your string.",true);
}
```

Calling this method is as simple as calling any other method.

```
var samples = FillTheseValues();
Console.WriteLine($"Int is: {samples.a}");
Console.WriteLine($"String is: {samples.b}");
Console.WriteLine($"Boolean is: {samples.c}");
```

Discards with Tuples

Following up on the `SplitNames()` example, suppose you know that you need only the first and last names and don't care about the first. By providing variable names for the values you want returned and filling in the unneeded values with an underscore (_) placeholder, you can refine the return value like this:

```
var (first, _, last) = SplitNames("Philip F Japikse");
Console.WriteLine($"{first}:{last}");
```

The middle name value of the tuple is discarded.

Tuples in method signatures:

The RockPaperScissors method signature could also be written to take in a tuple, like this:

```
static string RockPaperScissors2(
  (string first, string second) value)
{
  return value switch
  {
```