

---

# FONDAMENTI DI PROGRAMMAZIONE

-  
AR/VR AND GAME  
DEVELOPER

-  
Lesson 2

Fabio Cesarato

---

# SET-UP ENVIRONMENT

---

---

# Setting Up the Development Environment



You can develop C# applications using Visual Studio or Visual Studio Code.



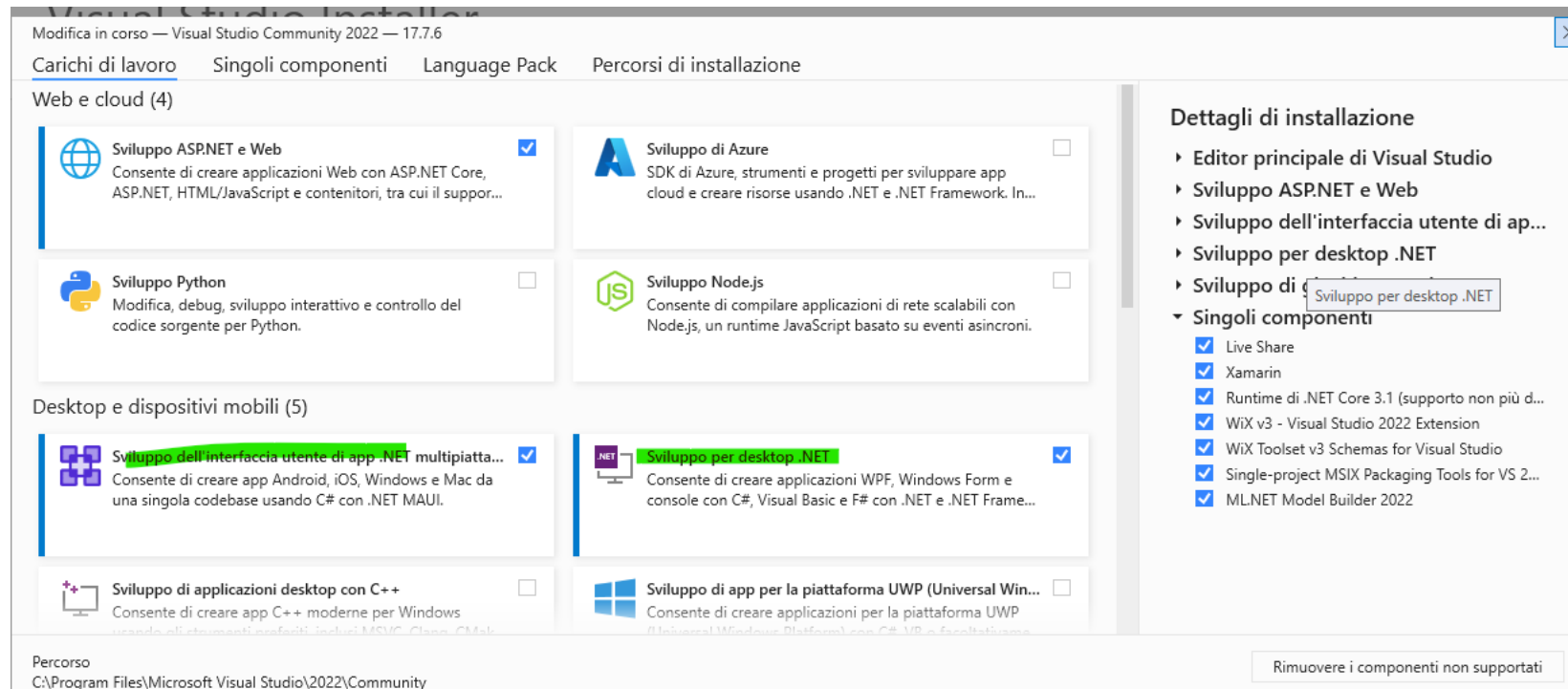
Visual Studio Community Edition is a free option for beginners.



Download and install the development environment to get started.

# Set-up for tests

<https://visualstudio.microsoft.com/it/vs/community/>





# Set-up for tests

Modifica in corso — Visual Studio Community 2022 — 17.7.6


Carichi di lavoro Singoli componenti Language Pack Percorsi di installazione


### Giochi (2)


 **Sviluppo di giochi con Unity** ☒  
Consente di creare giochi 2D e 3D con Unity, un potente ambiente di sviluppo multiplatforma.


 **Sviluppo di giochi con C++** ☐  
Sfruttare tutte le funzionalità di C++ per compilare giochi professionali basati su DirectX, Unreal o Cocos2d.


### Altri set di strumenti (5)

 **Elaborazione ed archiviazione dati** ☐  
Consente di connettere, sviluppare e testare soluzioni dati con SQL Server, Azure Data Lake o Hadoop.

 **Applicazioni analitiche e di analisi scientifica dei dati** ☐  
Linguaggi e strumenti per la creazione di applicazioni di data science, tra cui Python e F#.

 **Sviluppo di estensioni di Visual Studio** ☐  
Consente di creare componenti aggiuntivi ed estensioni per Visual Studio, inclusi nuovi comandi, analizzatori del...

 **Sviluppo per Office/SharePoint** ☐  
Consente di creare componenti aggiuntivi per Office e SharePoint, soluzioni SharePoint e componenti aggiuntivi...

 **Linux e sviluppo e incorporato con C++** ☐  
Crea ed esegui il debug di applicazioni eseguite in un...

Percorso  
C:\Program Files\Microsoft Visual Studio\2022\Community

---

# DEBUG





---

??

Admiral Grace Hopper, who worked at Harvard University in the 1940s. One of her colleagues found a moth impeding the operation of one of the university's computers, she told them they were **debugging the system**.



---

# Debug class

Provides a set of methods and properties that help debug your code.

Create code more robust **without impacting the performance and code size** of your shipping product.

Write methods in the following variations:

- Write
- WriteLine

```
Debug.WriteLine("Write a new line");  
Debug.Write("Write same line");
```



---

# Console class

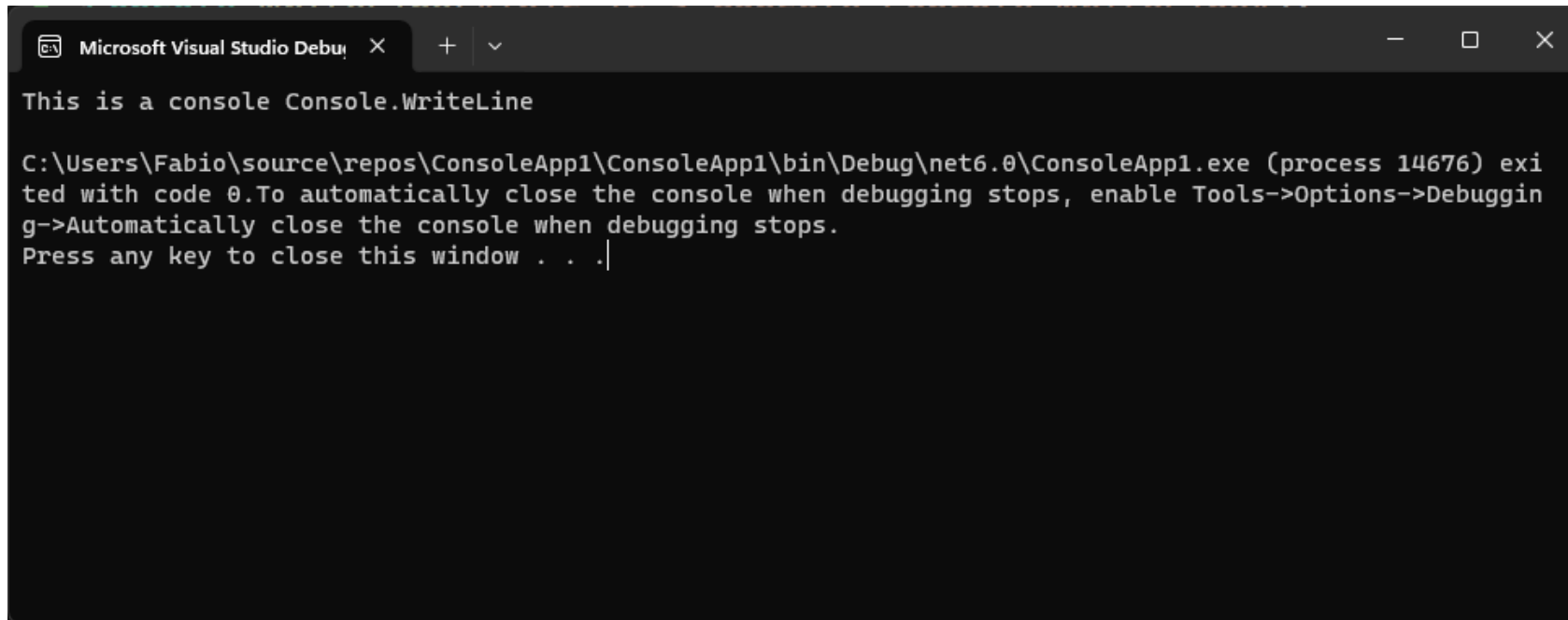
Represents the standard input, output, and error streams for console applications

- The **Write** method overloads convert an instance of a value type, an array of characters, or a set of objects to a formatted or unformatted string, and then write that string to the console.
- A parallel set of **WriteLine** method overloads output the same string as the Write overloads but also add a line termination string.
- The overloads of the ReadKey method read an individual character.
- The **ReadLine** method reads an entire line of input.

---

# Console class

```
Console.WriteLine("This is a console Console.WriteLine");
```



The screenshot shows a Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The output text is as follows:

```
This is a console Console.WriteLine  
  
C:\Users\Fabio\source\repos\ConsoleApp1\ConsoleApp1\bin\Debug\net6.0\ConsoleApp1.exe (process 14676) exited with code 0. To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .|
```

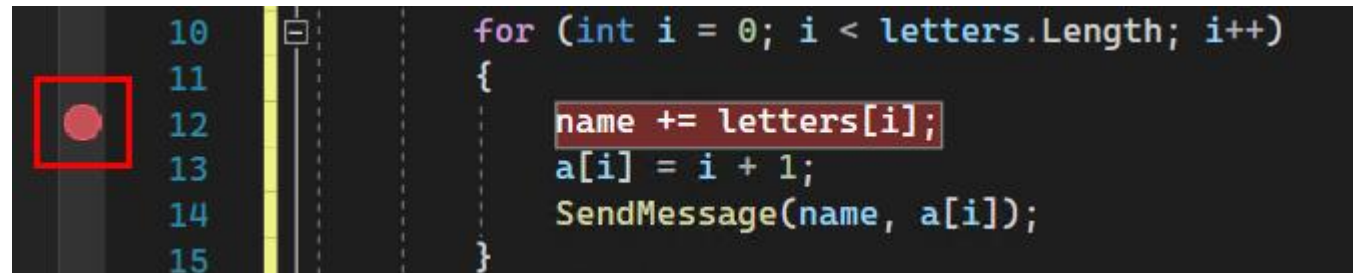


---

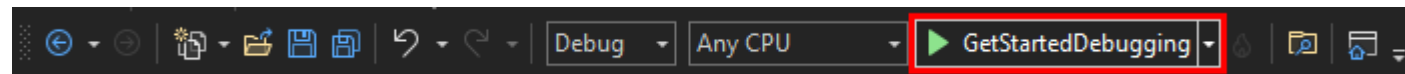
# Visual Studio - Debug

Example to follow: [microsoft learn](#)

Set breakpoints



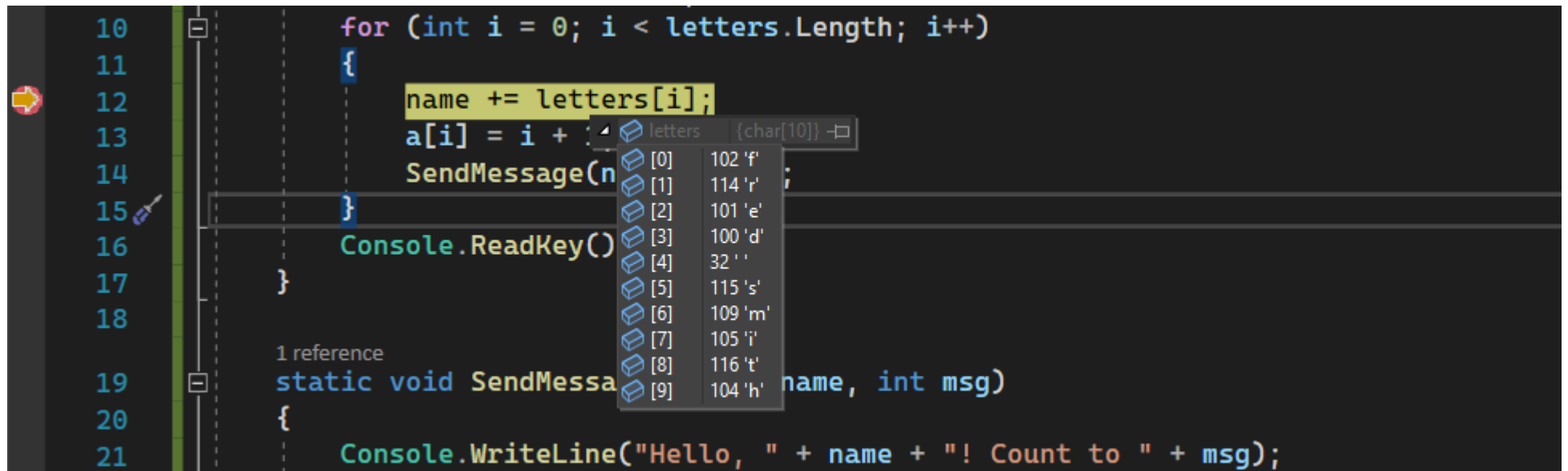
Start the debugger (F5)



---

# Visual Studio - Debug

Inspect data



```
10 for (int i = 0; i < letters.Length; i++)
11 {
12     name += letters[i];
13     a[i] = i + 1;
14     SendMessage(n
15 }
16 Console.ReadKey()
17 }
18
19 1 reference
20 static void SendMessa
21 {
22     Console.WriteLine("Hello, " + name + "! Count to " + msg);
```

	letters	{char[10]}
[0]	102	'f'
[1]	114	'r'
[2]	101	'e'
[3]	100	'd'
[4]	32	' '
[5]	115	's'
[6]	109	'm'
[7]	105	'i'
[8]	116	't'
[9]	104	'h'

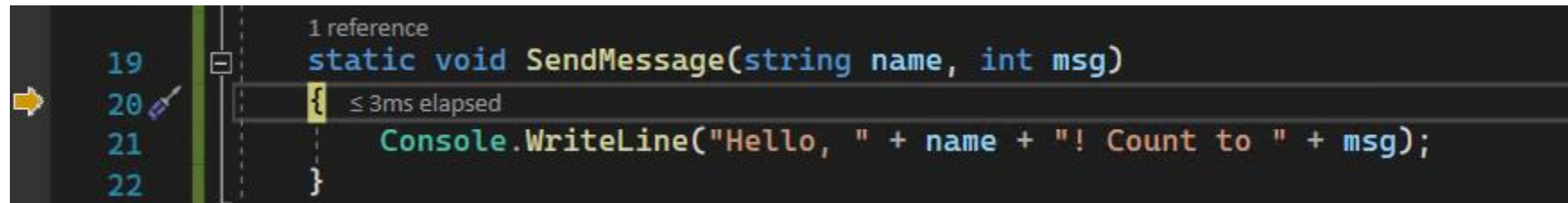
Step Over (F10): advance the debugger to the next statement

---

# Visual Studio - Debug

Step Into (F11): advances into a method

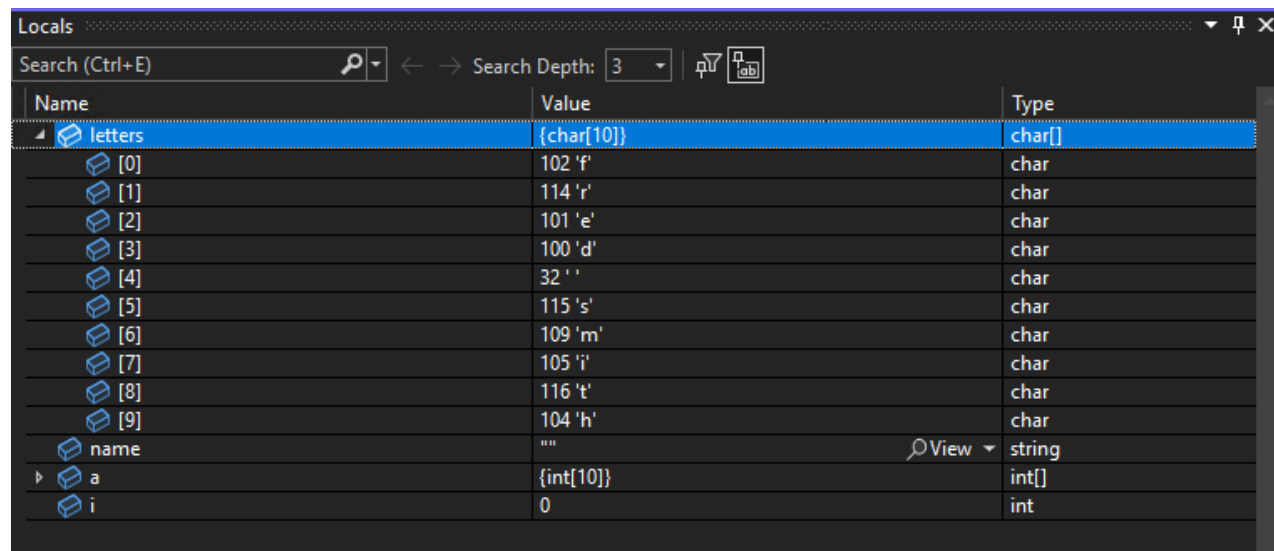
F11 helps you examine the execution flow of your code in more depth. To step into a method from a method call, select F11.



```
19  
20  
21  
22  
1 reference  
static void SendMessage(string name, int msg)  
{  
    Console.WriteLine("Hello, " + name + "! Count to " + msg);  
}
```

# Visual Studio - Debug

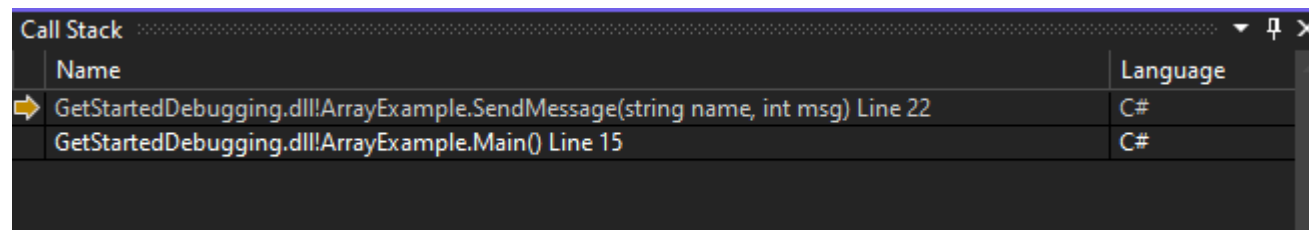
The Autos and **Locals** windows show variable values while you're debugging. The windows are only available during a debug session. The Autos window shows variables used on the current line that the debugger is at and the preceding line. The Locals window shows variables defined in the local scope, which is usually the current function or method.



---

# Visual Studio - Debug

**Call Stack:** understand the **execution flow** of your app, by showing the order in which methods and functions are getting called.



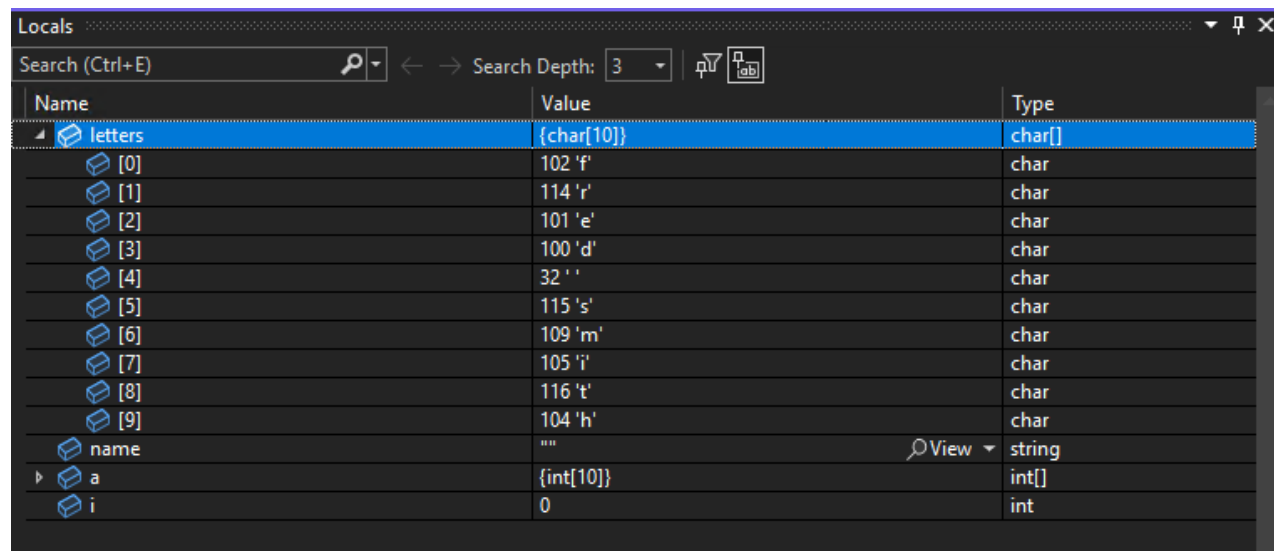
Stop Debugging





# Visual Studio - Debug

The Autos and **Locals** windows show variable values while you're debugging. The windows are only available during a debug session. The Autos window shows variables used on the current line that the debugger is at and the preceding line. The Locals window shows variables defined in the local scope, which is usually the current function or method.



---

# Unity - Debug

Logs a message to the Unity Console.

```
Debug.Log("Log");  
  
Debug.LogWarning("Warning");  
  
Debug.LogError("Error");
```



---

# Final Example

```
int a = 10;  
int b = 20;  
  
// Sum  
// Multiply  
// Divide  
// Print
```

```
int Sum(int a, int b)
{
    return a + b;
}

int Multiplication(int a, int b)
{
    return a * b;
}

double Division(double a, double b)
{
    return a / b;
}

int a = 10;
int b = 20;
// Sum
int sum = Sum(a, b);
// Multiply
int multiplication = Multiplication(a, b);
// Division - implicit conversion
double division = Division(a, b);
// Print
Console.WriteLine(sum);
Console.WriteLine(multiplication);
Console.WriteLine(division);
```



---

# DATA TYPES

# VALUE TYPES

---

# Value Types

- variable of a value type contains an instance of the type
- variables of value types directly contain their data
- value types, each variable has its own copy of the data, and it's not possible for operations on one variable to affect the other
- on assignment, passing an argument to a method, and returning a method result, variable values are copied. In the case of value-type variables, the corresponding type instances are copied.

---

# Enumeration

- An enumeration type (or enum type) is a **value type** defined by a set of named constants of the underlying integral numeric type.
- To define an enumeration type, use the **enum** keyword and specify the names of enum members

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```



---

# Enumeration

- By default, the associated constant values of enum members are of type int; they start with zero and increase by one following the definition text order. You can explicitly specify any other integral numeric type as an underlying type of an enumeration type.
- Can also explicitly specify the associated constant values

```
enum ErrorCode
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

---

# DATA TYPES REFERENCE TYPES - INTRO



---

# null

- The **null** keyword is a literal that represents a null reference, one that does not refer to any object.
- null is the default value of reference-type variables.
- Ordinary value types cannot be null, except for nullable value types.



---

# Nullable value types

- A nullable value type **T?** represents all values of its underlying value type T and an additional null value. For example, you can assign any of the following three values to a `bool?` variable: `true`, `false`, or `null`. An underlying value type T cannot be a nullable value type itself.
- Any nullable value type is an instance of the generic **System.Nullable<T>** structure. You can refer to a nullable value type with an underlying type T in any of the following interchangeable forms: `Nullable<T>` or `T?`.

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value
type:
int?[] arr = new int?[10];
```

---

# Nullable value types

Read-only properties to examine and get a value of a nullable value type variable:

- `Nullable<T>.HasValue` indicates whether an instance of a nullable value type has a value of its underlying type.
- `Nullable<T>.Value` gets the value of an underlying type if `HasValue` is true. If `HasValue` is false, the `Value` property throws an `InvalidOperationException`.

```
int? b = 10;
if (b.HasValue == true)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}

int? c = null;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
```

---

# Nullable value types - Example

```
Nullable<int> b = null;
if (b.HasValue == true)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}

b = 10;
if (b == null)
{
    Console.WriteLine("b does not have a value");
}
else
{
    Console.WriteLine($"b is {b.Value}");
}
```

# METHODS

Secant  
Lines

Tangent  
Line

$x+h$

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x) = \lim_{h \rightarrow 0} \frac{(x+h)^2 - x^2}{h}$$

$$= \lim_{h \rightarrow 0} \frac{x^2 + 2xh + h^2 - x^2}{h}$$

$$= \lim_{h \rightarrow 0} \frac{2xh + h^2}{h}$$

$$\begin{aligned} &= \lim_{h \rightarrow 0} \frac{h}{h(\sqrt{x+h} + \sqrt{x})} \\ &= \lim_{h \rightarrow 0} \frac{1}{\sqrt{x+h} + \sqrt{x}} \\ &= \frac{1}{2\sqrt{x}} \end{aligned}$$

$$\begin{aligned} f(x) &= \lim_{\Delta x \rightarrow 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \\ f(a) &= \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h} \end{aligned}$$

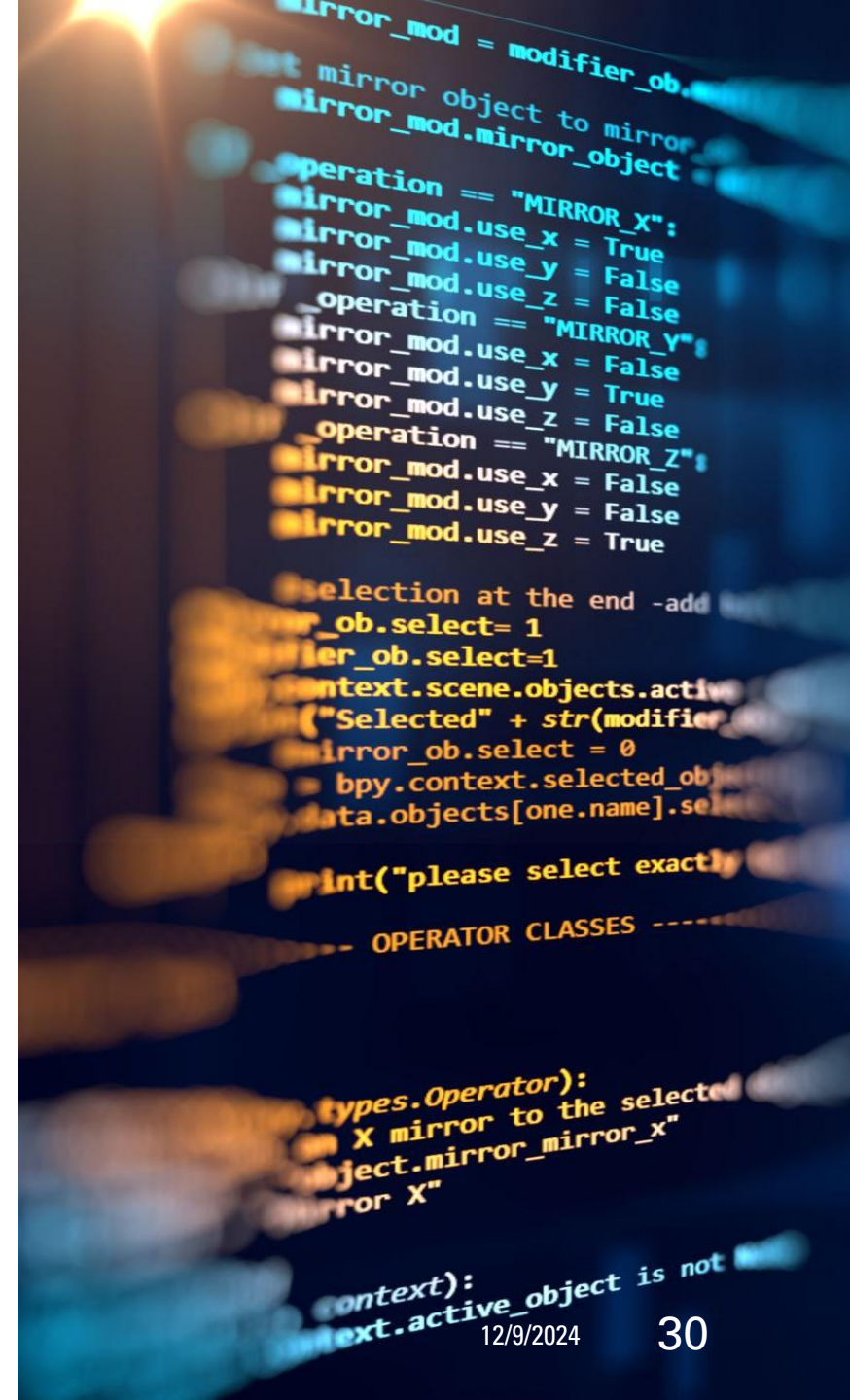


---

# Method

A method

- is a code block that contains a series of statements
- is a block of code which only runs when it is called
- you can pass data => **parameters**
- every executed instruction is performed in the **context** of a method



---

# Method

- Methods **can return a value** by including the return type in the method signature. Methods can return any data type, or they can return nothing at all.
- Using **void** as the return type means the method only performs operations and **doesn't return a value**.
- The **return type must always be specified** before the method name.

```
void VoidMethod(string stringParameter)  
  
bool BoolMethod(int intParameter)
```

---

# Method

- Methods can return a value by specifying the **return data type**, or **void** for no return value
- The **return** keyword can be used with variables, literals, and expressions
- The value returned from a method must match the specified return type
- Data returned from methods can be captured and used by the caller of the method

```
double total = 0;
double minimumSpend = 30.00;

double[] items = {15.97, 3.50, 12.25, 22.99, 10.98};
double[] discounts = {0.30, 0.00, 0.10, 0.20, 0.50};

for (int i = 0; i < items.Length; i++)
{
    total += GetDiscountedPrice(i);
}

total -= TotalMeetsMinimum() ? 5.00 : 0.00;

Console.WriteLine($"Total: ${FormatDecimal(total)}");

double GetDiscountedPrice(int itemIndex)
{
    return items[itemIndex] * (1 - discounts[itemIndex]);
}

bool TotalMeetsMinimum()
{
    return total >= minimumSpend;
}

string FormatDecimal(double input)
{
    return input.ToString().Substring(0, 5);
}
```

```
void DoStuff()
{
    double total = 0;

    double[] items = { 15.97, 3.50, 12.25, 22.99, 10.98 };
    double[] discounts = { 0.30, 0.00, 0.10, 0.20, 0.50 };

    for (int i = 0; i < items.Length; i++)
    {
        double itemFullPriceArg = items[i];
        double itemDiscountArg = discounts[i];
        double discountedPrice = GetDiscountedPrice(itemFullPriceArg, itemDiscountArg);
        total = total + discountedPrice;
    }

    PrintValue(total);
}

void PrintValue(double total)
{
    Console.WriteLine($"Edited Total: ${FormatDecimal(total)}");
}

double GetDiscountedPrice(double itemFullPrice, double itemDiscount)
{
    double result = itemFullPrice * (1 - itemDiscount);
    return result;
}

string FormatDecimal(double input)
{
    return input.ToString();
}

DoStuff();
```

---

# Method invocation

- Calling a method: after the object name (if you're calling an instance method) or the type name (if you're calling a static method), add a period, the name of the method, and parentheses. Arguments are listed within the parentheses and are separated by commas.
- The method definition specifies the names and types of any parameters that are required. When a caller invokes the method, it provides concrete values, called **arguments**, for each parameter. The arguments must be compatible with the parameter type, but the argument name, if one is used in the calling code, doesn't have to be the same as the parameter named defined in the method.

---

# Passing parameters by value

## Copy of the object

instead of the object itself is passed to the method.

Therefore, changes to the object in the called method have **no effect on the original object** when control returns to the caller.

```
using System;

public class ByValExample
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
```



---

# Passing parameters by reference

**Pass by reference means passing access to the variable to the method.**

**change the value of an argument** in a method and want to reflect that change when control returns to the calling method.

To pass a parameter by reference, you use the `ref` or `out` keyword.

---

# Passing parameters by reference - ref



- **ref** means the method can read or write the value of the argument.
- The argument must be initialized before calling the method. The method can assign a new value to the parameter, but isn't required to do so.
- An argument for a ref parameter must include the ref modifier.
- The ref keyword can't be used on the first argument of an extension method when the argument isn't a struct, or a generic type not constrained to be a struct.

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

---

# Passing parameters by reference - out



- **out** means the method sets the value of the argument.
- out: The calling method isn't required to initialize the argument before calling the method. The method must assign a value to the parameter.
- An argument for an out parameter must include the out modifier.
- The out keyword can't be used on the first argument of an extension method.

```
int initializeInMethod;  
OutArgExample(out initializeInMethod);  
Console.WriteLine(initializeInMethod);  
// value is now 44  
  
void OutArgExample(out int number)  
{  
    number = 44;  
}
```

---

# Passing parameters by reference - out



```
string stringValue = "test"
int number;

bool success = int.TryParse(stringValue, out number);
if (success)
{
    Console.WriteLine($"Converted '{stringValue}' to {number}.");
}
else
{
    Console.WriteLine($"Attempted conversion of '{stringValue}' failed.");
}
```

---

# Passing parameters - objects

When an **object** of a reference type is passed to a method, a **reference to the object is passed**. That is, the method receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, **even if you pass the object by value**.

```
public class SampleRefType
{
    public int value;
}

public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

---

# Optional arguments - Default Parameter

- can omit arguments for optional parameters
- a default value must be one of the following types of expressions:
  - a constant expression;
  - an expression of the form **new ValType()**, where ValType is a value type, such as an enum or a struct;
  - an expression of the form **default(ValType)**, where ValType is a value type.

```
public void ExampleMethod(int required, string optionalStr = "default string", int optionalInt = 10)

void Example(ExampleClass exampleDefault = null)
```

---

# Optional arguments - Default Parameter - Example

```
void ExampleMethod(int required, string optionalStr = "default string", int optionalInt = 10)
{
    Console.WriteLine(required);
    Console.WriteLine(optionalStr);
    Console.WriteLine(optionalInt);
}

ExampleMethod(1);
Console.WriteLine("----");
ExampleMethod(2, "test");
Console.WriteLine("----");
ExampleMethod(3, "test2", 42);
```

---

# Study Material

## INTRO

<https://dotnet.microsoft.com/en-us/learn/csharp>

<https://www.youtube.com/playlist?list=PLdo4fOcmZ0oVxKLOCHpiUWun7vIJJvUiN>

## CODE

<https://learn.microsoft.com/it-it/collections/yz26f8y64n7k07>

<https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>

## DEBUG

<https://learn.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-debugger>

<https://learn.microsoft.com/en-us/visualstudio/debugger/debugging-absolute-beginners>

## DESIGN PATTERNS

<https://www.dofactory.com/net/design-patterns>

---

