LẬP TRÌNH C# - SOLOLEARN

Student : Green Wolf

Date : 27 / 12 / 2020

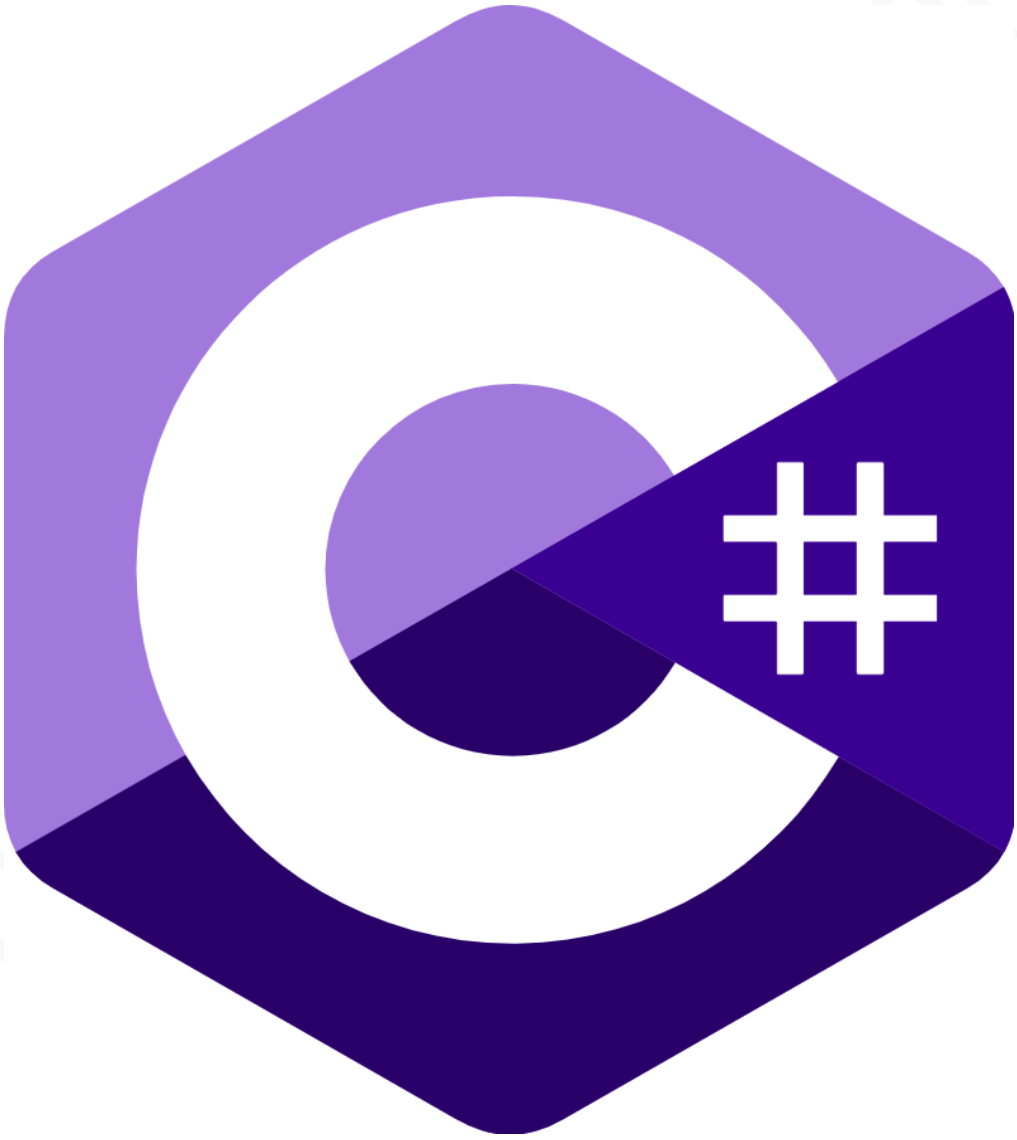


Table of Contents

Title **1**

Table of Contents **2**

1. Basic Concepts **5**

1. What is C#? 5
2. Variables 6
3. Your First C# Program 9
4. Printing Text 13
5. Getting User Input 14
6. Comments 16
7. The var Keyword 18
8. Constants 19
9. Arithmetic Operators 20
10. Assignment & Increment Operators 23
11. Module 1 Quiz 27

2. Conditionals and Loops **30**

1. The if-else Statement 30
2. The switch Statement 36
3. The while Loop 41
4. The for Loop 45
5. The do-while Loop 48
6. break and continue 50
7. Logical Operators 52
8. The Conditional Operator 56
9. Basic Calculator 57
10. Module 2 Quiz 60

3. Methods **63**

1. Introduction to Methods 63
2. Method Parameters 67
3. Multiple Parameters 70
4. Optional & Named Arguments 72

5. Passing Arguments 76
6. Method Overloading 80
7. Recursion 83
8. Making a Pyramid 84
9. Module 3 Quiz 87

4. Classes & Objects 90

1. Classes & Objects 90
2. Value & Reference Types 92
3. Class Example 94
4. Encapsulation 97
5. Constructors 101
6. Properties 105
7. Module 4 Quiz 112

5. Arrays & Strings 117

1. Arrays 117
2. Using Arrays in Loops 120
3. Multidimensional Arrays 124
4. Jagged Arrays 127
5. Array Properties & Methods 128
6. Working with Strings 131
7. Module 5 Quiz 134

6. More On Classes 137

1. Destructors 137

7. Inheritance & Polymorphism 137

8. Structs, Enums, Exceptions & Files 137

9. Generics 137

Dictionary 138

Noun 138
Verb 141
Adjective 143
Other 144

End 144

1. Basic Concepts

1. What is C#?

Welcome to C#

C# is an elegant object-oriented language that enables developers to build a variety of secure and robust applications that run on the **.NET Framework.**

You can use C# to create Windows applications, Web services, mobile applications, client-server applications, database applications, and much, much more.

You will learn more about these concepts in the upcoming lessons!

C# applications run:

```
on the .NET Framework
```

```
using Java
```

```
only on Linux
```

The .NET Framework

The .NET Framework consists of the **Common Language Runtime (CLR)** and the .NET Framework **class library**.

The **CLR** is the foundation of the .NET Framework. It manages code at execution time, providing core services such as memory management, code accuracy, and many other aspects of your code.

The **class library** is a collection of classes, interfaces, and value types that enable you to accomplish a range of common programming tasks, such as data collection, file access, and working with text.

C# programs use the .NET Framework class library extensively to do common tasks and provide various functionalities.

*These concepts might seem complex, but for now just remember that applications written in C# use the **.NET Framework** and its components.*

Which one is NOT part of the .NET Framework?

```
.Net Framework Class Library
```

```
Operation System
```

```
Common Language Runtime
```

2. Variables

Variables

Programs typically use data to perform tasks.

Creating a **variable** reserves a memory location, or a space in memory, for storing values. It is called **variable** because the information stored in that location can be changed when the program is running.

To use a variable, it must first be declared by specifying the **name** and **data type**.

A variable name, also called an **identifier**, can contain letters, numbers and the underscore character (_) and must start with a letter or underscore.

Although the name of a variable can be any set of letters and numbers, the best identifier is descriptive of the data it will contain. This is very important in order to create clear, understandable and readable code!

*For example, **firstName** and **lastName** are good descriptive variable names, while **abc** and **xyz** are not.*

Which is a valid C# variable name?

```
1Star
```

```
#PersonName#
```

```
Bad_Var
```

Variable Types

A **data type** defines the information that can be stored in a variable, the size of needed memory and the operations that can be performed with the variable.

For example, to store an integer value (a whole number) in a variable, use the **int** keyword:

```
int myAge;
```

The code above declares a variable named **myAge** of type **integer**.

*A line of code that completes an action is called a statement. Each statement in C# must end with a **semicolon ";"**.*

You can assign the value of a variable when you declare it:

```
int myAge = 18;
```

or later in your code:

```
int myAge;
myAge = 18;
```

Remember that you need to declare the variable before using it.

Fill in the blanks to declare a variable named num of type integer and assign 42 to it.

```
int num;
num = 42;
```

Built-in Data Types

There are a number of built-in data types in C#. The most common are:

int - integer.

float - floating point number.

double - double-precision version of float.

char - a single character.

bool - Boolean that can have only one of two values: True or False.

string - a sequence of characters.

The statements below use C# data types:

```
int x = 42;
double pi = 3.14;
char y = 'Z';
bool isOnline = true;
string firstName = "David";
```

*Note that **char** values are assigned using single quotes and **string** values require double quotes.*

You will learn how to perform different operations with variables in the upcoming lessons!

Drag and drop the correct data types from the options below.

```
bool a = false;
double b = 4.22;
string c = "Hi";
int d = 11;
```

3. Your First C# Program

Your First C# Program

You can run, save, and share your C# codes on our **Code Playground**, without installing any additional software.

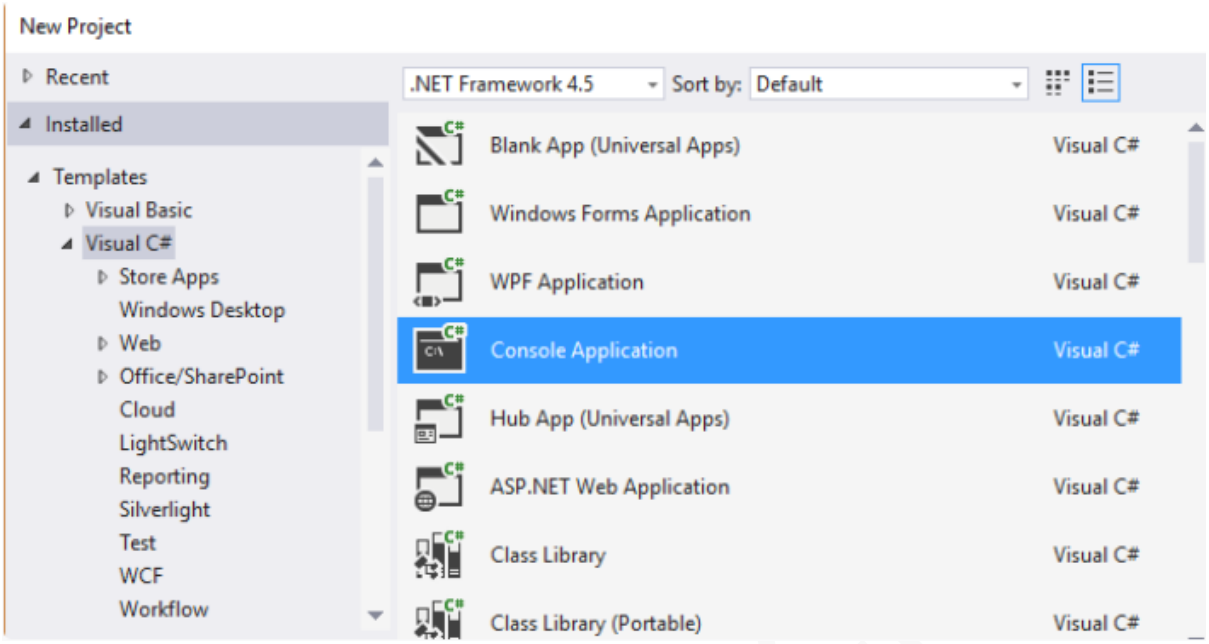
Reference this lesson if you need to install the software on your computer.

To create a C# program, you need to install an integrated development environment (IDE) with coding and debugging tools.

We will be using **Visual Studio Community Edition**, which is available to download for free.

After installing it, choose the default configuration.

Next, click **File->New->Project** and then choose **Console Application** as shown below:



Enter a name for your Project and click OK.

Console application *uses a text-only interface. We chose this type of application to focus on learning the fundamentals of C#.*

What is the name of the IDE used to create C# programs?

- Visual Studio
- 3D Studio
- CStudio
- Visual Maya

Visual Studio will automatically generate some code for your project:

```
using System;
```

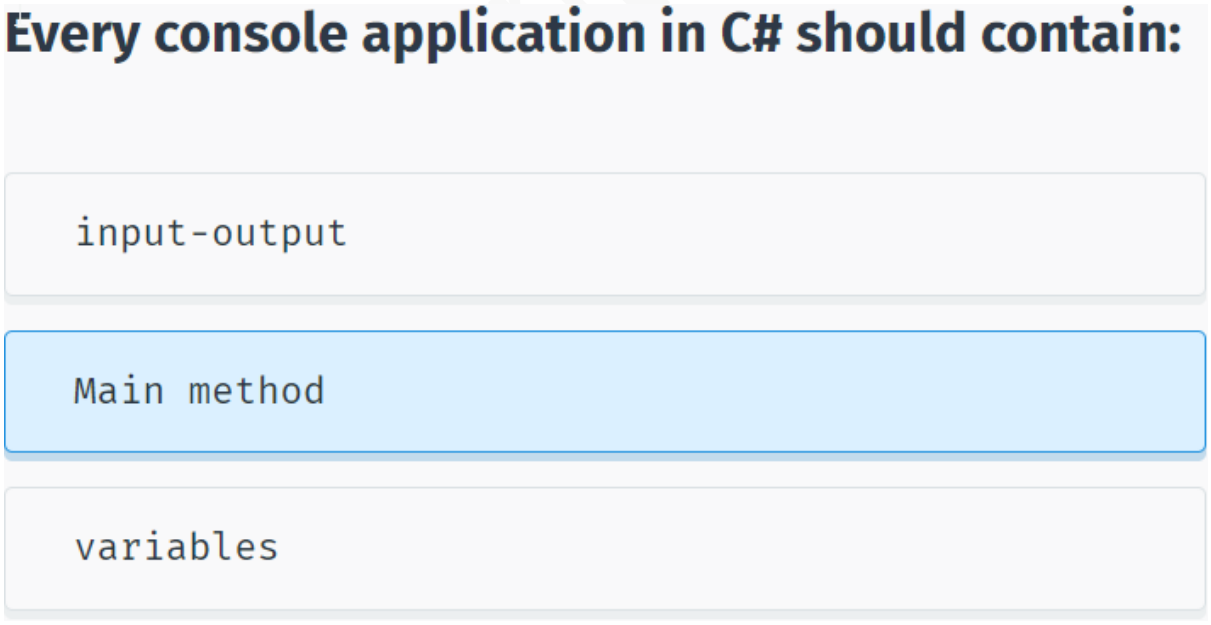
```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

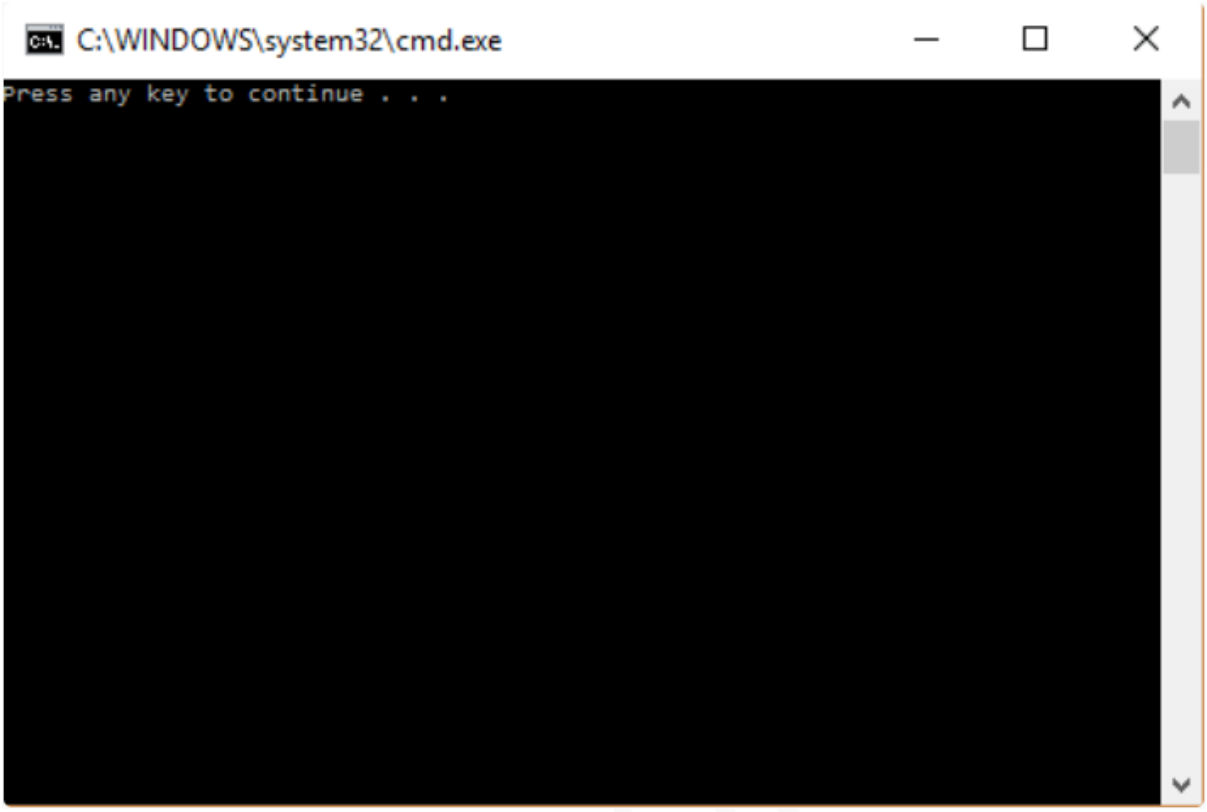
You will learn what each of the statements does in the upcoming lessons.

For now, remember that every C# console application must contain a **method (a function) named Main**. Main is the starting point of every application, i.e. the point where our program starts execution from.

We will learn about classes, methods, arguments, and namespaces in the upcoming lessons.



To run your program, press **Ctrl+F5**. You will see the following screen:



This is a console window. As we did not have any statements in our **Main** method, the program just produces a general message. Pressing any key will close the console.

Congratulations, you just created your first C# program.

Which type of application uses a text-only interface?

- Mobile Application
- Windows Application
- Console Application

4. Printing Text

Displaying Output

Most applications require some **input** from the user and give **output** as a result.

To display text to the console window you use the **Console.Write** or **Console.WriteLine** methods. The difference between these two is that **Console.WriteLine** is followed by a line terminator, which moves the cursor to the next line after the text output.

The program below will display Hello World! to the console window:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

*Note the **parentheses** after the **WriteLine** method. This is the way to pass data, or arguments, to methods. In our case **WriteLine** is the method and we pass "Hello World!" to it as an argument. String arguments must be enclosed in quotation marks.*

Drag and drop from the options below to display "Learning C#".

```
Console. WriteLine ( "Learning C#" );
```

We can display variable values to the console window:

```
int x = 89;
Console.WriteLine(x);//89
```

To display a **formatted string**, use the following syntax:

```
int x = 10;
double y = 20;
Console.WriteLine("x = {0}; y = {1}", x, y);
//x = 10; y = 20
```

As you can see, the value of **x** replaced **{0}** and the value of **y** replaced **{1}**.

You can have as many variable placeholders as you need. (i.e.: {3}, {4}, etc.).

What is the output of this code?

```
int a = 4;
int b = 2;
Console.Write(a);
Console.Write(b);
```

```
42
```

5. Getting User Input

User Input

You can also prompt the user to enter data and then use the **Console.ReadLine** method to assign the input to a string variable.

The following example asks the user for a name and then displays a message that includes the input:

```
string yourName;
Console.WriteLine("What is your name?");
yourName = Console.ReadLine();//->Green
Console.WriteLine("Hello {0}", yourName);
//What is your name?
//Green
```

The **Console.ReadLine** method waits for user input and then assigns it to the variable. The next statement displays a formatted string containing Hello with the user input. For example, if you enter David, the output will be Hello David.

*Note the empty parentheses in the **ReadLine** method. This means that it does not take any arguments.*

Drag and drop from the options below to take user input and store it in the temp variable:

```
string temp;

temp = [ Console ] . [ ReadLine ] ();
```

The **Console.ReadLine()** method returns a **string** value.

If you are expecting another type of value (such as int or double), the entered data must be converted to that type.

This can be done using the **Convert.ToXXX** methods, where XXX is the .NET name of the type that we want to convert to. For example, methods include **Convert.ToDouble** and **Convert.ToBoolean**.

For integer conversion, there are three alternatives available based on the bit size of the integer: **Convert.ToInt16**, **Convert.ToInt32** and **Convert.ToInt64**. The default int type in C# is 32-bit.

Let's create a program that takes an integer as input and displays it in a message:

```
int age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("You are {0} years old", age);
```

*If, in the program above, a non-integer value is entered (for example, letters), the **Convert** will fail and cause an error.*

Drag and drop from the options below to make the following program work:

```
double n;
string x = "77";
n = [ Convert ].[ ToDouble ](x);
```

6. Comments

Comments

Comments are explanatory statements that you can include in a program to benefit the reader of your code.

The compiler ignores everything that appears in the comment, so none of that information affects the result.

A comment beginning with two slashes (//) is called a single-line comment. The slashes tell the compiler to ignore everything that follows, until the end of the line.

```
// Prints Hello
Console.WriteLine("Hello");
```

When you run this code, Hello will be displayed to the screen. The // Prints Hello line is a comment and will not appear as output.

What is the output of this code?

```
int x = 8;
// x = 3;
Console.WriteLine(x);
```

- error
- 8
- 3

Multi-Line Comments

Comments that require multiple lines begin with /* and end with */ at the end of the comment block.

You can place them on the same line or insert one or more lines between them.

```
/* Some long
comment text
*/
int x = 42;
Console.WriteLine(x);//42
```

Adding comments to your code is good programming practice. It facilitates a clear understanding of the code for you and for others who read it.

Fill in the blanks to make the text a comment.

```
  /*   Declaring an integer
    and printing it   */
int x = 42;
Console.WriteLine(x);
```

7. The var Keyword

The var Keyword

A variable can be explicitly declared with its **type** before it is used.

Alternatively, C# provides a handy function to enable the compiler to determine the type of the variable automatically based on the expression it is assigned to.

The var keyword is used for those scenarios:

```
var num = 15;
```

The code above makes the compiler determine the type of the variable. Since the value assigned to the variable is an integer, the variable will be declared as an integer automatically.

What is the type of the temp variable?
var temp = 14.55;

```
Double
```

Variables declared using the var keyword are called **implicitly typed** variables.

Implicitly typed variables must be initialized with a value.

For example, the following program will cause an error:

```
var num;
num = 42;->error!
```

*Although it is easy and convenient to declare variables using the **var** keyword, overuse can harm the readability of your code. Best practice is to explicitly declare variables.*

What is the output of this code?

```
var n1;
n1 = true;
Console.WriteLine(n1);
```

1

true

error

8. Constants

Constants

Constants store a value that cannot be changed from their initial assignment.

To declare a constant, use the **const** modifier.

For example:

```
const double PI = 3.14;
```

The value of const PI cannot be changed during program execution.

For example, an assignment statement later in the program will cause an error:

```
const double PI = 3.14;
PI = 8; ->error
```

*Constants **must** be initialized with a value when declared.*

Fill in the blank to make the variable num a constant.

```
const int num=2;
```

9. Arithmetic Operators

Operators

An **operator** is a symbol that performs mathematical or logical manipulations.

Arithmetic Operators

C# supports the following arithmetic operators:

Operator	Symbol	Form
Addition	+	x + y
Subtraction	-	x - y
Multiplication	*	x * y
Division	/	x / y
Modulus	%	x % y

For example:

```
int x = 10;
int y = 4;
Console.WriteLine(x-y);//6
```

*Tap **Try It Yourself** to play around with the code!*

Fill in the blanks to display the result of the multiplication of x and y.

```
int x = 42;

int y = 7;

int z = x  *  y;

Console.WriteLine(  z  );
```

Division

The division operator (/) divides the first operand by the second. If the operands are both integers, any remainder is dropped in order to return an integer value.

Example:

```
int x = 10 / 4;
Console.WriteLine(x);//2
```

Division by 0 is undefined and will crash your program.

What is the output of this code?

```
int x = 16;
int y = 5;
Console.WriteLine(x/y);
```

3

Modulus

The modulus operator (%) is informally known as the remainder operator because it returns the remainder of an integer division.

For example:

```
int x = 25 % 7;
Console.WriteLine(x);//4
```

Which operator is used to determine the remainder?

- +
- *
- %

Operator Precedence

Operator **precedence** determines the grouping of terms in an expression, which affects how an expression is evaluated. Certain operators take higher precedence over others; for example, the multiplication operator has higher precedence than the addition operator.

For example:

```
int x = 4+3*2;
Console.WriteLine(x);//10
```

The program evaluates 3*2 first, and then adds the result to 4.

As in mathematics, using **parentheses** alters operator precedence.

```
int x = (4 + 3) *2;
Console.WriteLine(x);//14
```

The operations within parentheses are performed first. If there are parenthetical expressions nested within one another, the expression within the innermost parentheses is evaluated first.

If none of the expressions are in parentheses, multiplicative (multiplication, division, modulus) operators will be evaluated before additive (addition, subtraction) operators. Operators of equal precedence are evaluated from left to right.

Fill in the missing parentheses to have x equal 15.

```
int x =    (    2 + 3    )    * 3;

Console.WriteLine(x);
```

10. Assignment & Increment Operators

Assignment Operators

The = **assignment** operator assigns the value on the right side of the operator to the variable on the left side.

C# also provides **compound assignment operators** that perform an operation and an assignment in one statement.

For example:

```
int x = 42;
x += 2; // equivalent to x = x + 2
Console.WriteLine(x);//44
x -= 6; // equivalent to x = x - 6
Console.WriteLine(x);//38
```

What is the alternative for x = x + 5?

```
x = y + 5;
```

```
x -= 4;
```

```
x += 5;
```

The same shorthand syntax applies to the multiplication, division, and modulus operators.

```
x *= 8; // equivalent to x = x * 8
x /= 5; // equivalent to x = x / 5
x %= 2; // equivalent to x = x % 2
```

The same shorthand syntax applies to the multiplication, division, and modulus operators.

Fill in the missing part of the following code to divide x by 3 using the shorthand division operator.

```
int x = 42;

x [ / ] = [ 3 ] ;
```

Increment Operator

The **increment** operator is used to increase an integer's value by one, and is a commonly used C# operator.

```
x++; //equivalent to x = x + 1
```

For example:

```
int x = 10;
x++;
Console.WriteLine(x);//11
```

The increment operator is used to increase an integer's value by one.

What is the output of this code?

```
int x = 6;
x++;
Console.WriteLine(x);
```

7

Prefix & Postfix Forms

The increment operator has two forms, **prefix** and **postfix**

```
++x; //prefix
x++; //postfix
```

Prefix increments the value, and then proceeds with the expression.

Postfix evaluates the expression and then performs the incrementing.

Prefix example:

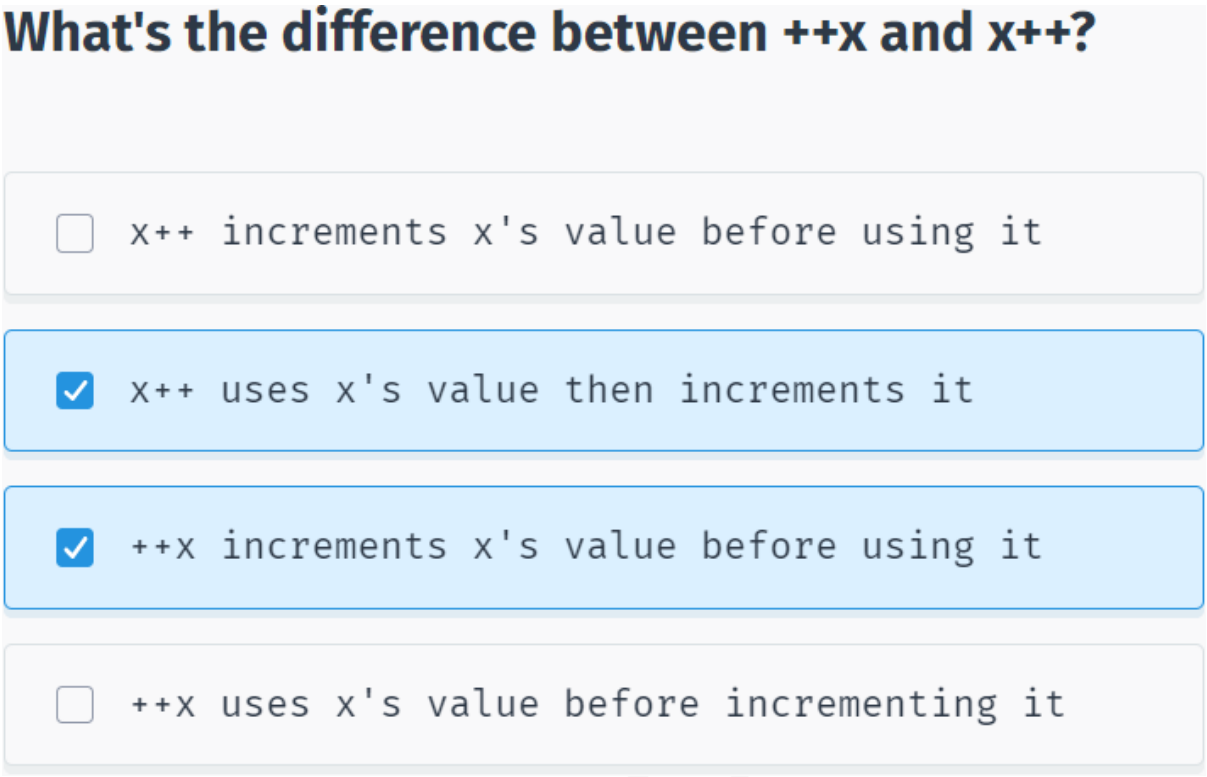
```
int x = 3;
int y = ++x;
//x is 4; y is 4
```

Postfix example:

```
int x = 3;
int y = x++;
//x is 4; y is 3
```

*The **prefix** example increments the value of x, and then assigns it to y.*

*The **postfix** example assigns the value of x to y, and then increments x.*



Decrement Operator

The **decrement** operator (--) works in much the same way as the increment operator, but instead of increasing the value, it decreases it by one.

```
--x; // prefix
x--; // postfix
```

The decrement operator (--) works in much the same way as the increment operator.

Fill in the missing operator to decrease the value of x by one.

```
int x = 42;

x --;

Console.WriteLine(x);
```

11. Module 1 Quiz

Rearrange the code to create a valid program.

```
static void Main(string[] args) {
string msg = "Hello";
Console.WriteLine(msg);
}
```

Drag and drop from the options below to display the value of x to the screen.

```
static void Main (string[] args)
{
    int x = (4 + 3) * 2;
    Console . WriteLine (x);
}
```

What is the output of this code?

```
int a = 4;
int b = 6;
b = a++;
Console.WriteLine(++b);
```

5

Fill in the blanks to declare two variables of type int and display their sum to the screen.

```
int x = 8;

    int     y = 15;

Console.WriteLine(x  +  y);
```

What is the output of this code?

```
int x = 15;
int y = 6;
x %= y;
Console.WriteLine(x);
```

3

2. Conditionals and Loops

1. The if-else Statement

The if Statement

The **if** statement is a conditional statement that executes a block of code when a condition is true.

The general form of the if statement is:

```
if (condition)
{
    // Execute this code when condition is true
}
```

The condition can be any expression that returns true or false.

For example:

```
int x = 8;
int y = 3;
if (x > y)
{
     Console.WriteLine("x is greater than y");
     //x is greater than y
}
```

The code above will evaluate the condition **x > y**. If it is true, the code inside the if block will execute.

When only one line of code is in the if block, the curly braces can be omitted.

For example:

```
if (x > y)
Console.WriteLine("x is greater than y");
```

Fill in the blanks to display Welcome to the screen when age is greater than 16.

```
int age = 24;

    if (age > 16)

{

    Console.WriteLine("Welcome");

}
```

Relational Operators

Use **relational operators** to evaluate conditions. In addition to the less than (<) and greater than (>) operators, the following operators are available:

Operator	Description	Example	
>=	Greater than or equal to	7 >= 4	True
<=	Less than or equal to	7 <= 4	False
==	Equal to	7 == 4	False
!=	Not equal to	7 != 4	True

Example:

```
int a=7, b=7;
if (a == b) {
     Console.WriteLine("Equal");
     //Equal
}
```

Which is the correct operator for equality testing?

```
==
```

```
>=
```

```
!=
```

```
<=
```

The else Clause

An optional **else** clause can be specified to execute a block of code when the condition in the **if** statement evaluates to **false**.

Syntax:

```
if (condition)
{
   //statements
}
else
{
   //statements
}
```

For example:

```
int mark = 85;
if (mark < 50)
{
     Console.WriteLine("You failed.");
}
else
```

```
{
      Console.WriteLine("You passed.");
}
```

Fill in the blanks to find the larger of two variables.

```
int a = 42;

int b = 88;

 if (a > b)

{

   Console.WriteLine(a);

}

   else

{

   Console.WriteLine( b );

}
```

Nested if Statements

You can also include, or **nest**, if statements within another if statement.

For example:

```
int mark = 100;
if (mark >= 50) {
```

```
      Console.WriteLine("You passed.");
      if (mark == 100) {
          Console.WriteLine("Perfect!");
      }
else {
      Console.WriteLine("You failed.");
}
```

You can nest an unlimited number of if-else statements.

For example:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int age = 17;
            if (age > 14) {
                if(age > 18) {
                    Console.WriteLine("Adult");
                }
                else {
                    Console.WriteLine("Teenager");
                }
            }
            else {
                if (age > 0) {
                    Console.WriteLine("Child");
                }
                else {
                    Console.WriteLine("Something's wrong");
                }
            }
        }
```

```
    }
}
```

*Remember that all **else** clauses must have corresponding **if** statements.*

What is the output of this code?

```
int a = 8;
int b = ++a;
if(a > 5)
  b -= 3;
else
  b = 9;
Console.WriteLine(b);
```

6

The if-else if Statement

The **if-else if** statement can be used to decide among three or more actions.

For example:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 33;
```

```
            if (x == 8) {
                Console.WriteLine("Value of x is 8");
            }
            else if (x == 18) {
                Console.WriteLine("Value of x is 18");
            }
            else if (x == 33) {
                Console.WriteLine("Value of x is 33");
            }
            else {
                Console.WriteLine("No match");
            }
        }
    }
}
```

*Remember, that an **if** can have zero or more **else if**'s and they must come before the last **else**, which is optional.*

*Once an **else if** succeeds, none of the remaining **else if**'s or **else** clause will be tested.*

How many nested if statements can an if statement contain?

- Only two
- As many as you want
- None

2. The switch Statement

switch

The **switch** statement provides a more elegant way to test a variable for equality against a list of values.

Each value is called a **case**, and the variable being switched on is checked for each switch case.

For example:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int num = 3;
            switch (num)
            {
                case 1:
                    Console.WriteLine("one");
                    break;
                case 2:
                    Console.WriteLine("two");
                    break;
                case 3:
                    Console.WriteLine("three");
                    break;
            }
        }
    }
}
```

Each **case** represents a value to be checked, followed by a colon, and the statements to get executed if that case is matched.

*A **switch** statement can include any number of **cases**. However, no two case labels may contain the same constant value.*

*The **break**; statement that ends each **case** will be covered shortly.*

Fill in the blanks to create a valid switch statement.

```
int x = 33;

switch (x) {
  case 8:
    Console.WriteLine("Value is 8");
    break;
  case 18 :
    Console.WriteLine("Value is 18");
    break;
  case 33:
    Console.WriteLine("Value is 33");
    break;
}
```

The default Case

In a switch statement, the optional **default** case is executed when none of the previous cases match.

Example:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int age = 88;
            switch (age) {
                case 16:
                    Console.WriteLine("Too young");
                    break;
                case 42:
                    Console.WriteLine("Adult");
                    break;
                case 70:
                    Console.WriteLine("Senior");
                    break;
                default:
                    Console.WriteLine("The default case");
                    break;
            }
        }
    }
}
```

*The **default** code executes when none of the cases matches the switch expression.*

Drag and drop from the options below to test the x variable with a switch statement.

```
switch (x) {
  case 10:
    Console.WriteLine("Ten");
    break;
  case 20:
    Console.WriteLine("Twenty");
    break;
  default:
    Console.WriteLine("No match");.
    break;
}
```

The break Statement

The role of the **break** statement is to terminate the **switch** statement.

Without it, execution continues past the matching **case** statements and falls through to the next case statements, even when the case labels don't match the switch variable.

This behavior is called **fallthrough** and modern C# compilers will not compile such code. All case and default code must end with a **break** statement.

*The **break** statement can also be used to break out of a loop. You will learn about loops in the coming lessons.*

What would occur if we forget to include a break statement at the end of case code?

- nothing
- "break" will be printed
- compile error

3. The while Loop

while

A **while** loop repeatedly executes a block of code as long as a given condition is **true**.

For example, the following code displays the numbers 1 through 5:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int num = 1;
            while(num < 6)
            {
```

```
                Console.WriteLine(num);
                num++;
            }
        }
    }
}
```

The example above declares a variable equal to 1 (int num = 1). The **while** loop checks the condition (num < 6) and, if **true**, executes the statements in its body, which increment the value of **num** by one, before checking the loop condition again.

After the 5th iteration, **num** equals 6, the condition evaluates to **false**, and the loop stops running.

*The **loop body** is the block of statements within curly braces.*

Fill in the blanks to display the value of x to the screen three times.

```
int x = 42;

int num = 0;

  while (num < 3) {

  Console.WriteLine( x );

  num++;

}
```

The while Loop

The compound arithmetic operators can be used to further control the number of times a loop runs.

For example:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int num = 1;
            while(num < 6)
            {
                Console.WriteLine(num);
                num+=2;
            }
        }
    }
}
```

*Without a statement that eventually evaluates the loop condition to **false**, the loop will continue indefinitely.*

Fill in the blanks to increment the value of num by 2 to display only even values.

```
int num = 0;

while ( num  < 100)

{

  Console.WriteLine(num);

  num  +  =  2  ;

}
```

We can shorten the previous example, by incrementing the value of **num** right in the condition:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int num = 0;
            while(++num < 6)
                Console.WriteLine(num);
        }
```

```
    }
}
```

*What do you think, is there a difference between **while(num++ < 6)** and **while(++num < 6)**?*

*Yes! The loop **while(++num < 6)** will execute 5 times, because pre-increment increases the value of x before checking the num < 6 condition, while post-increment will check the condition before increasing the value of num, making **while(num++ < 6)** execute 6 times.*

How many times will the following loop execute?

```
int x = 1;
while (x++ < 5)
{
   if (x % 2 == 0)
      x += 2;
}
```

2

4. The for Loop

The for Loop

A **for** loop executes a set of statements a specific number of times, and has the syntax:

```
for ( init; condition; increment ) {
  statement(s);
}
```

A counter is declared once in **init**.

Next, the **condition** evaluates the value of the counter and the body of the loop is executed if the condition is **true**.

After loop execution, the **increment** statement updates the counter, also called the loop control variable.

The condition is again evaluated, and the loop body repeats, only stopping when the condition becomes **false**.

For example:

```
for (int x = 10; x < 15; x++)
{
     Console.WriteLine("Value of x: {0}", x);
}
```

*Note the **semicolons** in the syntax.*

Drag and drop from the options below to create a general for loop:

```
for ( int x = 5 ;  x < 10  ;  x++  )
{
  Console.WriteLine(x);
}
```

Compound arithmetic operators can be used to further control loop iterations.

For example:

```
for (int x = 0; x < 10; x+=3)
{
     Console.WriteLine(x);
}
```

You can also decrement the counter:

```
for (int x = 10; x > 0; x-=2)
{
     Console.WriteLine(x);
}
```

Fill in the blanks to print the EVEN values from 0 to 100 using a for loop:

```
for (int x = 0; x < 100; x+= 2 )
{
  Console.WriteLine( x );
}
```

The **init** and **increment** statements may be left out, if not needed, but remember that the semicolons are mandatory.

For example, the init can be left out:

```
int x = 10;
for ( ; x > 0; x -= 3)
{
     Console.WriteLine(x);
}
```

You can have the increment statement in the for loop body:

```
int x = 10;
for ( ; x > 0 ; )
{
     Console.WriteLine(x);
     x -= 3;
}
```

***for (; ;) {}** is an infinite loop.*

How many times will this loop run?

```
int x = 1;
for ( ; x < 7; )
{
  x+=2;
}
```

3

5. The do-while Loop

do-while

A **do-while** loop is similar to a **while** loop, except that a **do-while** loop is guaranteed to execute at least one time.

For example:

```
int a = 0;
do
{
    Console.WriteLine(a);
    a++;
} while (a < 5);
```

*Note the **semicolon** after the while statement.*

Fill in the blanks to create a valid loop.

```
int x = 0;
do {
  Console.WriteLine(x);
  x+=2;
} while (x < 10) ;
```

do-while vs. while

If the condition of the **do-while** loop evaluates to **false**, the statements in the **do** will still run once:

```
int x = 42;
do {
    Console.WriteLine(x);
    x++;
} while(x < 10);
```

*The **do-while** loop executes the statements at least once, and then tests the condition.*

*The **while** loop executes the statement only after testing condition.*

What is the output of this code?

```
int a = 2;
do {
  a+=3;
} while(a < 4);
Console.Write(a);
```

5

6. break and continue

break

We saw the use of **break** in the switch statement.

Another use of **break** is in loops: When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program execution moves on to the next statement following the loop body.

For example:

```
int num = 0;
while (num < 20)
{
    if (num == 5)
        break;

    Console.WriteLine(num);
    num++;
}
```

*If you are using nested loops (i.e., one loop inside another loop), the **break** statement will stop the execution of the innermost loop and start executing the next line of code after the block.*

What is the largest number that will be printed by this code?

```
for (int x = 1; x < 8; x++) {
  if (x > 5)
    break;
  Console.WriteLine(x);
}
```

5

continue

The **continue** statement is similar to the **break** statement, but instead of terminating the loop entirely, it skips the current iteration of the loop and continues with the next iteration.

For example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5)
        continue;

    Console.WriteLine(i);
}
```

*As you can see, number 5 is not printed, as the **continue** statement skips the remaining statements of that iteration of the loop.*

Fill in the blanks to print only even numbers.

```
for(int x=0; x<99; x++)  {
   if (x%2 != 0)
      continue;
   Console.WriteLine(x);
}
```

7. Logical Operators

Logical Operators

Logical operators are used to join multiple expressions and return **true** or **false**.

Operator	Name of Operator	Form
&&	**AND** Operator	x && y
||	**OR** Operator	x || y
!	**NOT** Operator	! x

The **AND** operator (&&) works the following way:

Left Operand	Right Operand	Result
false	false	**false**
false	true	**false**
true	false	**false**
true	true	**true**

For example, if you wish to display text to the screen only if **age** is greater than 18 AND **money** is greater than 100:

```
int age = 42;
double money = 540;
if(age > 18 && money > 100) {
    Console.WriteLine("Welcome");
}
```

The AND operator was used to combine the two expressions.

*With the AND operator, both operands must be **true** for the entire expression to be **true**.*

The result of a&&b is true if:

Never

Both a and b are true

Both a and b are false

Either a or b is true

AND

You can join more than two conditions:

```
int age = 42;
int grade = 75;
if(age > 16 && age < 80 && grade > 50)
    Console.WriteLine("Hey there");
```

*The entire expression evaluates to **true** only if all of the conditions are **true**.*

How many && operators can be used in one if statement?

Two

Only one

As many as you want

The OR Operator

The **OR** operator (||) returns **true** if any one of its operands is **true**.

Left Operand	Right Operand	Result
false	false	**false**
false	true	**true**
true	false	**true**
true	true	**true**

For example:

```
int age = 18;
int score = 85;
```

```
if (age > 20 || score > 50) {
    Console.WriteLine("Welcome");
}
```

*You can join any number of logical **OR** statements you want.*

*In addition, multiple **OR** and **AND** statements may be joined together.*

What is the output of this code?

```
int x = 5; int y = 12;
if(x>10 || y/x > 1)
  Console.Write(y-x);
else
  Console.Write(y);
```

- 7
- 5
- 12

Logical NOT

The logical **NOT** (!) operator works with just a single operand, reversing its logical state. Thus, if a condition is **true**, the NOT operator makes it **false**, and vice versa.

Right Operand	Result
true	false
false	true

```
int age = 8;
```

```
if ( !(age > 16) ) {
    Console.Write("Your age is less than 16");
}
```

If a is true and b is false, what is the result of !(a&&b)?

- false
- undefined
- true

8. The Conditional Operator

The ? : Operator

Consider the following example:

```
int age = 42;
string msg;
if(age >= 18)
    msg = "Welcome";
else
    msg = "Sorry";

Console.WriteLine(msg);
```

The code above checks the value of the **age** variable and displays the corresponding message to the screen.

This can be done in a more elegant and shorter way by using the **?: operator**, which has the following form:

```
Exp1 ? Exp2 : Exp3;
```

The ?: operator works the following way: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

So, the example above can be replaced by the following:

```
int age = 42;
string msg;
msg = (age >= 18) ? "Welcome" : "Sorry";
Console.WriteLine(msg);
```

What is the value of x after this code?

```
int x = 5;
int y = 3;
x = (x > y) ? y : x;
```

3

9. Basic Calculator

Basic Calculator

Now let's create a simple project that repeatedly asks the user to enter two values and then displays their sum, until the user enters exit.

We start with a **do-while** loop that asks the user for input and calculates the **sum**:

```
do {
  Console.Write("x = ");
  int x = Convert.ToInt32(Console.ReadLine());

  Console.Write("y = ");
  int y = Convert.ToInt32(Console.ReadLine());

  int sum = x+y;
  Console.WriteLine("Result: {0}", sum);
}
while(true);
```

This code will ask for user input infinitely. Now we need to handle the "exit".

If the user enters a non-integer value, the program will crash from a conversion error. We will learn how to handle errors like that in the coming modules.

How many times would this loop run?

```
do { }
while(false);
```

- none
- infinite
- one

If the user enters "exit" as the value of x, the program should quit the loop. To do this, we can use a **break** statement:

```
Console.Write("x = ");
string str = Console.ReadLine();
if (str == "exit")
  break;
int x = Convert.ToInt32(str);
```

Here we compare the input with the value "exit" and break the loop.

So the whole program looks like:

```
do {
  Console.Write("x = ");
  string str = Console.ReadLine();
  if (str == "exit")
    break;

  int x = Convert.ToInt32(str);

  Console.Write("y = ");
```

```
  int y = Convert.ToInt32(Console.ReadLine());

  int sum = x + y;
  Console.WriteLine("Result: {0}", sum);
}
while (true);
```

If the user enters "exit" as the value of x, the program should quit the loop.

Which of the following is used to take user input?

- Console.Write
- Convert.ToInt32
- Console.ReadLine

10. Module 2 Quiz

Fill in the blanks to print the value of x five times.

```
int x = 42;

int num = 0;

while(num < [ 5 ] ) {

    Console.WriteLine( [ x ] );

    [ num ] ++;

}
```

Drag and drop from the options below to create a valid finite for loop.

```
[ for ] ( [ int x=0 ] ; [ x<10 ] ; [ x++ ]
) {

  Console.WriteLine(x);

}
```

Select the correct statements about && and || operators.

- [x] (a && b) || c is true if c is true
- [ ] a && b is false if both a and b are true
- [ ] a && b is true if either a or b is true
- [x] a || b is true if either a or b is true

Fill in the blanks to calculate the sum of all whole numbers from 1 to 100.

```
int sum = 0;

for (int x=1;x<=100; x ++) {

    sum += x;

}

Console.WriteLine(sum);
```

What is the value of x after this code?

```
int x = 4; int y = 9;
x = (y%x != 0) ? y/x : y;
```

2

3. Methods

1. Introduction to Methods

What is a Method?

A **method** is a group of statements that perform a particular task.

In addition to the C# built-in methods, you may also define your own.

Methods have many advantages, including:

- Reusable code.
- Easy to test.
- Modifications to a method do not affect the calling program.
- One method can accept many different inputs.

*Every valid C# program has at least one method, the **Main** method.*

Every C# program starts with the method:

```
Main
```

Declaring Methods

To use a method, you need to **declare** the method and then **call** it.

Each method declaration includes:

- the return type
- the method name
- an optional list of parameters.

```
<return type> name(type1 par1, type2 par2, … , typeN parN)
{
    List of statements
}
```

For example, the following method has an int parameter and returns the number squared:

```
int Sqr(int x)
{
  int result = x*x;
  return result;
```

```
}
```

The **return** type of a method is declared before its name. In the example above, the return type is **int**, which indicates that the method returns an integer value. When a method returns a value, it must include a **return** statement. Methods that return a value are often used in assignment statements.

Occasionally, a method performs the desired operations without returning a value. Such methods have a return type **void**. In this case, the method cannot be called as part of an assignment statement.

***void** is a basic data type that defines a valueless state.*

If you do not want your method to return a value, you should use the return type:

```
void
```

Calling Methods

Parameters are optional; that is, you can have a method with no parameters.

As an example, let's define a method that does not return a value, and just prints a line of text to the screen.

```
static void SayHi()
{
  Console.WriteLine("Hello");
}
```

Our method, entitled **SayHi**, returns **void**, and has no parameters.

To execute a method, you simply call the method by using the name and any required arguments in a statement.

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

namespace SoloLearn
```

```
{
    class Program
    {
        static void sayHi()
        {
            Console.WriteLine("Hello");
        }
        static void Main(string[] args)
        {
            sayHi();
        }
    }
}
```

*The **static** keyword will be discussed later; it is used to make methods accessible in Main.*

Fill in the blanks to declare a method that does not return a value and displays "Welcome" to the screen:

```
static    void    Greet()
{
  Console.WriteLine("Welcome");
  }
```

You can call the same method multiple times:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SoloLearn
{
    class Program
    {
        static void SayHi()
        {
            Console.WriteLine("Hello");
        }
        static void Main(string[] args)
        {
            SayHi();
            SayHi();
            SayHi();
        }
    }
}
```

Drag and drop from the options below to declare a valid method and call it in Main:

```
static   void   Func()
{
  Console.Write("test");
}
static void Main(string[] args)
{
   Func   ();
}
```

2. Method Parameters

Parameters

Method declarations can define a list of **parameters** to work with.

Parameters are variables that accept the values passed into the method when called.

For example:

```
void Print(int x)
{
  Console.WriteLine(x);
}
```

This defines a method that takes one integer parameter and displays its value.

Parameters behave within the method similarly to other local variables. They are created upon entering the method and are destroyed upon exiting the method.

Fill in the blanks to declare a method that takes one integer parameter and then displays the value divided by 2.

```
void MyFunc( int x)
{
  int result = x / 2;
  Console.WriteLine( result );
}
```

Now you can call the method in Main and pass in the value for its parameters (also called **arguments**):

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

namespace SoloLearn
{
    class Program
    {
        static void Print(int x)
        {
```

```
            Console.WriteLine(x);
        }
        static void Main(string[] args)
        {
            Print(42);
        }
    }
}
```

The value 42 is passed to the method as an argument and is assigned to the formal parameter x.

Fill in the blanks to declare a method and call it from Main with the argument 88:

```
static void func(int x)
{
  Console.WriteLine(x/2);
}
static void Main(string[] args)
{
    func ( 88 );
}
```

You can pass different arguments to the same method as long as they are of the expected type.

For example:

```
static void Func(int x)
{
    Console.WriteLine(x*2);
}
static void Main(string[] args)
{
    Func(5);

    Func(12);

    Func(42);
}
```

How many times can you call a method with different arguments?

None

One

As many as you want

3. Multiple Parameters

Multiple Parameters

You can have as many parameters as needed for a method by separating them with **commas** in the definition.

Let's create a simple method that returns the sum of two parameters:

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

The **Sum** method takes two integers and returns their sum. This is why the return type of the method is **int**. Data **type** and **name** should be defined for each parameter.

*Methods return values using the **return** statement.*

Fill in the missing parts of the following code to define a method that returns an int value and has two parameters.

```
int Test(int a , int b)
{
 // some code
}
```

A method call with multiple parameters must separate arguments with **commas**.

For example, a call to **Sum** requires two arguments:

```
static int Sum(int x, int y)
{
    return x+y;
}
static void Main(string[] args)
{
    Console.WriteLine(Sum(8, 6));
}
```

In the call above, the return value was displayed to the console window. Alternatively, we can assign the return value to a variable, as in the code below:

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

```
}
static void Main(string[] args)
{
    int res = Sum(11, 42);
    Console.WriteLine(res);
}
```

*You can add as many parameters to a single method as you want. If you have multiple parameters, remember to separate them with **commas**, both when declaring them and when calling the method.*

Fill in the blanks to declare a method, which returns the largest value of its parameters:

```
int Max(int a [ , ] int b)
{
  if(a > b)
    [ return ] a;
  else
    return [ b ];
}
```

4. Optional & Named Arguments

Optional Arguments

When defining a method, you can specify a **default value** for optional parameters. Note that optional parameters must be defined after required

parameters. If corresponding arguments are missing when the method is called, the method uses the default values.

To do this, assign values to the parameters in the method definition, as shown in this example.

```
static int Pow(int x, int y=2)
{
  int result = 1;
  for (int i = 0; i < y; i++)
  {
    result *= x;
  }

  return result;
}
```

The Pow method assigns a default value of 2 to the y parameter. If we call the method without passing the value for the y parameter, the default value will be used.

```
static int Pow(int x, int y=2)
{
    int result = 1;
    for (int i = 0; i < y; i++)
    {
        result *= x;
    }
    return result;
}
static void Main(string[] args)
{
    Console.WriteLine(Pow(6));

    Console.WriteLine(Pow(3, 4));
}
```

As you can see, default parameter values can be used for calling the same method in different situations without requiring arguments for every parameter.

*Just remember, that you must have the parameters with default values at the **end** of the parameter list when defining the method.*

What is the output of this code?

```
static int Vol(int x, int y=3, int z=1) {
  return x*y*z;
}
static void Main(string[] args) {
  Console.WriteLine(Vol(2, 4));
}
```

8

Named Arguments

Named arguments free you from the need to remember the order of the parameters in a method call. Each argument can be specified by the matching parameter name.

For example, the following method calculates the area of a rectangle by its height and width:

```
static int Area(int h, int w)
{
  return h * w;
}
```

When calling the method, you can use the parameter names to provide the arguments in any order you like:

```
static int Area(int h, int w)
{
    return h * w;
}
static void Main(string[] args)
{
    int res = Area(w: 5, h: 8);
    Console.WriteLine(res);
```

```
}
```

*Named arguments use the **name** of the parameter followed by a **colon** and the **value**.*

Call the method using named arguments with the values 5 for "from", 99 for "to" and 2 for "step":

```
static int calc(int from, int to, int step=1) {
  int res=0;
  for(int i=from;i<to;i+=step) {
    res += i;
  }
  return res;
}
static void Main(string[] args)
{
  int res = calc ( step: 2, to: 99, from: 5);
  Console.WriteLine(res);
}
```

5. Passing Arguments

Passing Arguments

There are three ways to pass arguments to a method when the method is called: By **value**, By **reference**, and as **Output**.

By **value** copies the argument's value into the method's formal parameter. Here, we can make changes to the parameter within the method without having any effect on the argument.

*By default, C# uses call by **value** to pass arguments.*

The following example demonstrates by value:

```
static void Sqr(int x)
{
    x = x * x;
}
static void Main(string[] args)
{
    int a = 3;
    Sqr(a);

    Console.WriteLine(a);
}
```

In this case, **x** is the parameter of the **Sqr** method and **a** is the actual argument passed into the method.

*As you can see, the **Sqr** method does not change the original value of the variable, as it is passed by **value**, meaning that it operates on the **value**, not the actual variable.*

What is the output of this code?

```
static void Test(int x) {
  x = 8;
}
static void Main() {
  int a = 5;
  Test(a);
  Console.WriteLine(a);
}
```

5

Passing by Reference

Pass by **reference** copies an argument's memory address into the formal parameter. Inside the method, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

To pass the value by reference, the **ref** keyword is used in both the call and the method definition:

```
static void Sqr(ref int x)
{
    x = x * x;
}
static void Main(string[] args)
{
    int a = 3;
    Sqr(ref a);

    Console.WriteLine(a);
}
```

The **ref** keyword passes the memory address to the method parameter, which allows the method to operate on the actual variable.

*The **ref** keyword is used both when defining the method and when calling it.*

Fill in the blanks to create a method that swaps the values of its two arguments.

```
void Swap( ref int x, ref int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp ;
}
```

Passing by Output

Output parameters are similar to reference parameters, except that they transfer data out of the method rather than accept data in. They are defined using the **out** keyword.

The variable supplied for the output parameter need not be initialized since that value will not be used. Output parameters are particularly useful when you need to return multiple values from a method.

For example:

```
static void GetValues(out int x, out int y)
{
    x = 5;
```

```
    y = 42;
}
static void Main(string[] args)
{
    int a, b;
    GetValues(out a, out b);
    Console.WriteLine(a+" "+b);
}
```

Unlike the previous reference type example, where the value 3 was referred to the method, which changed its value to 9, output parameters get their value from the method (5 and 42 in the above example).

*Similar to the **ref** keyword, the **out** keyword is used both when defining the method and when calling it.*

Fill in the blanks to ask for user input in the method and return the value entered using output parameters.

```
static void Ask(    out    string name)
{
    name    = Console.ReadLine();
}
static void Main(string[] args)
{
  string nm;
  Ask(  out  nm);
}
```

6. Method Overloading

Overloading

Method **overloading** is when multiple methods have the **same name**, but **different parameters**.

For example, you might have a **Print** method that outputs its parameter to the console window:

```
void Print(int a)
{
```

```
  Console.WriteLine("Value: "+a);
}
```

The + operator is used to concatenate values. In this case, the value of a is joined to the text "Value: ".

This method accepts an integer **argument** only.

Overloading it will make it available for other types, such as **double**

```
void Print(double a)
{
  Console.WriteLine("Value: "+a);
}
```

*Now, the same **Print** method name will work for both integers and doubles.*

Method overloading means:

printing values

Same parameters, different method names

Same method name, different parameters

When overloading methods, the definitions of the methods must differ from each other by the types and/or number of parameters.

When there are overloaded methods, the method called is based on the arguments. An **integer** argument will call the method implementation that accepts an **integer** parameter. A **double** argument will call the implementation that accepts a **double** parameter. Multiple arguments will call the implementation that accepts the same number of arguments.

```
static void Print(int a) {
    Console.WriteLine("Value: " + a);
}
static void Print(double a) {
    Console.WriteLine("Value: " + a);
}
```

```
static void Print(string label, double a) {
    Console.WriteLine(label + a);
}
static void Main(string[] args)
{
    Print(11);
    Print(4.13);
    Print("Average: ", 7.57);
}
```

You cannot overload method declarations that differ only by return type.

*The following declaration results in an **error**.*

***int** PrintName(int a) { }*

***float** PrintName(int b) { }*

***double** PrintName(int c) { }*

What is the output of this code?

```
static void Print(int a) {
  Console.WriteLine(a*a);
}
static void Print(double a) {
  Console.WriteLine(a+a);
}
static void Main(string[] args) {
  Print(3);
}
```

9

7. Recursion

Recursion

A **recursive** method is a method that calls itself.

One of the classic tasks that can be solved easily by recursion is calculating the **factorial** of a number.

In mathematics, the term **factorial** refers to the product of all positive integers that are less than or equal to a specific non-negative integer (n). The factorial of n is denoted as **n!**

For example:

```
4! = 4 * 3 * 2 * 1 = 24
```

A recursive method is a method that calls itself.

What is the factorial of 5?

```
24
```

```
120
```

As you can see, a factorial can be thought of as repeatedly calculating num * num-1 until you reach 1.

Based on this solution, let's define our method:

```
static int Fact(int num) {
  if (num == 1) {
    return 1;
  }
  return num * Fact(num - 1);
}
```

In the **Fact** recursive method, the **if** statement defines the exit condition, a base case that requires no recursion. In this case, when **num** equals one, the solution is simply to return 1 (the factorial of one is one).

The recursive call is placed after the exit condition and returns **num** multiplied by the factorial of n-1.

For example, if you call the **Fact** method with the argument 4, it will execute as follows:

return 4*Fact(3), which is 4*3*Fact(2), which is 4*3*2*Fact(1), which is 4*3*2*1.

Now we can call our Fact method from Main:

```
static int Fact(int num) {
    if (num == 1) {
        return 1;
    }
    return num * Fact(num - 1);
}
static void Main(string[] args)
{
    Console.WriteLine(Fact(6));
}
```

The factorial method calls itself, and then continues to do so, until the argument equals 1. The exit condition prevents the method from calling itself indefinitely.

What prevents the recursive method to call itself forever?

- the Main method
- the exit condition
- the static keyword

8. Making a Pyramid

Making a Pyramid

Now, let's create a method that will display a pyramid of any height to the console window using star (*) symbols.

Based on this description, a parameter will be defined to reflect the number of rows for the pyramid.

So, let's start by declaring the method:

```
static void DrawPyramid(int n)
{
   //some code will go here
}
```

DrawPyramid does not need to return a value and takes an integer parameter n.

In programming, the step by step logic required for the solution to a problem is called an **algorithm**. The algorithm for MakePyramid is:

1. The first row should contain one star at the top center of the pyramid. The center is calculated based on the number of rows in the pyramid.

2. Each row after the first should contain an odd number of stars (1, 3, 5, etc.), until the number of rows is reached.

Based on the algorithm, the code will use for loops to display spaces and stars for each row:

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

namespace SoloLearn
{
  class Program
  {
    static void DrawPyramid(int n)
    {
      for (int i=1; i<=n; i++)
      {
        for (int j=i; j<=n; j++)
        {
          Console.Write("  ");
        }
        for (int k=1; k<=2*i-1; k++)
        {
          Console.Write("*"+" ");
        }
```

```
            Console.WriteLine();
        }
    }
    static void Main(string[] args)
    {
        DrawPyramid(5);
    }
  }
}
```

The first **for** loop that iterates through each row of the pyramid contains two **for** loops.

The first inner loop displays the spaces needed before the first star symbol. The second inner loop displays the required number of stars for each row, which is calculated based on the formula (2*i-1) where i is the current row.

The final **Console.WriteLine();** statement moves the cursor to the next row.

*Now, if we call the **DrawPyramid** method, it will display a pyramid having the number of rows we pass to the method.*

How many loops can you nest within each other?

None

Any

One

9. Module 3 Quiz

Every C# program has the method:

```
Console
```

```
Main
```

Fill in the blanks to create a method that calculates and returns the sum of its parameters.

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

Fill in the blanks to declare a method that has two int parameters with default values 6 and 8, respectively, and displays their product to the screen. Call the method in Main using named arguments.

```
static void Test(int x = 6, int y= 8 ) {
  Console.WriteLine(x*y);
}
static void Main(string[] args)
{
  Test(x:7, y: 11);
}
```

If a method does not return any value, you should use the return type:

```
void
```

What is the output of this code?

```
static int Test(out int x, int y=4) {
  x = 6;
  return x * y;
}
static void Main(string[] args) {
  int a;
  int z = Test(out a);
  Console.WriteLine(a + z);
}
```

```
30
```

4. Classes & Objects

1. Classes & Objects

Classes

As we have seen in the previous modules, built-in data types are used to store a single value in a declared variable. For example, **int x** stores an integer value in a variable named **x**.

In object-oriented programming, a **class** is a data type that defines a set of variables and methods for a declared **object**.

For example, if you were to create a program that manages bank accounts, a **BankAccount** class could be used to declare an object that would have all the properties and methods needed for managing an individual bank account, such as a **balance** variable and **Deposit** and **Withdrawal** methods.

A class is like a **blueprint**. It defines the data and behavior for a type. A class definition starts with the keyword **class** followed by the class name. The class body contains the data and actions enclosed by curly braces.

```
class BankAccount
{
  //variables, methods, etc.
}
```

*The class defines a data type for objects, but it is not an object itself. An **object** is a concrete entity based on a class, and is sometimes referred to as an **instance of a class**.*

To use custom data types, you need to define

- an int variable
- a class
- methods

Objects

Just as a built-in data type is used to declare multiple variables, a class can be used to declare multiple **objects**. As an analogy, in preparation for a new building, the architect designs a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings.

Programming works in the same fashion. We define (design) a class that is the blueprint for creating objects.

In programming, the term **type** is used to refer to a class **name**: We're creating an object of a particular **type**.

Once we've written the class, we can create objects based on that class. Creating an object is called **instantiation**.

An object is called an instance of a class.

The process of creating objects is called:

```
creation
```

```
objection
```

```
classification
```

```
instantiation
```

Each object has its own characteristics. Just as a person is distinguished by name, age, and gender, an object has its own set of values that differentiate it from another object of the same type.

The characteristics of an object are called **properties**.

Values of these properties describe the current state of an object. For example, a Person (an object of the class Person) can be 30 years old, male, and named Antonio.

Objects aren't always representative of just physical characteristics.

For example, a programming object can represent a date, a time, and a bank account. A bank account is not tangible; you can't see it or touch it, but it's still a well-defined object because it has its own properties.

Let's move on and see how to create your own custom classes and objects!

Which of the following determines the current state of an object?

- the name of the object
- classes
- properties

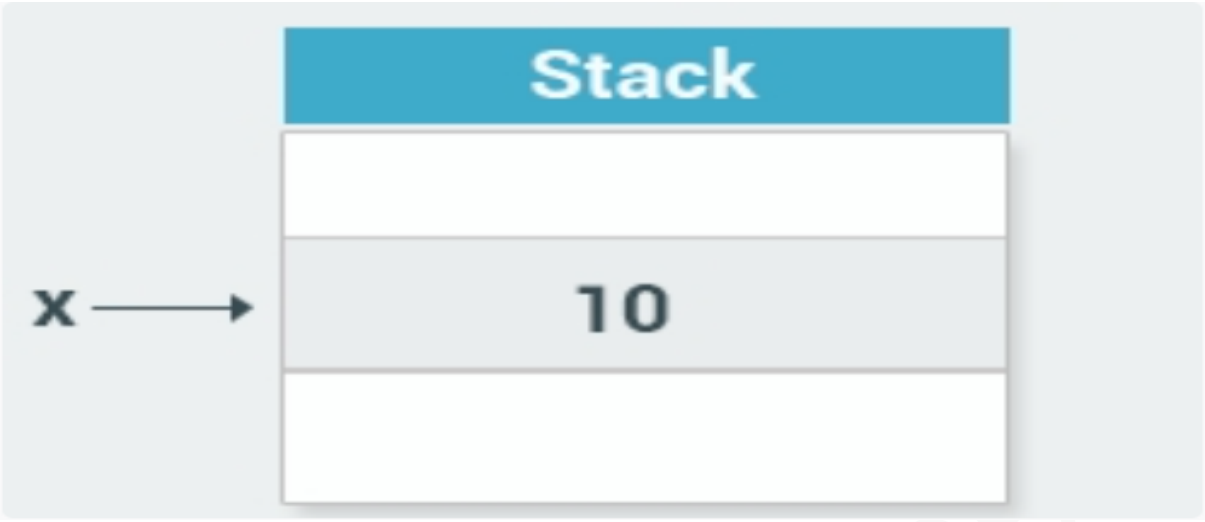
2. Value & Reference Types

Value Types

C# has two ways of storing data: by **reference** and by **value**.

The built-in data types, such as int and double, are used to declare variables that are **value types**. Their value is stored in memory in a location called the **stack**.

For example, the declaration and assignment statement **int x = 10**; can be thought of as:



*The value of the variable x is now stored on the **stack**.*

The area in memory that stores the contents of a value type is called:

stack

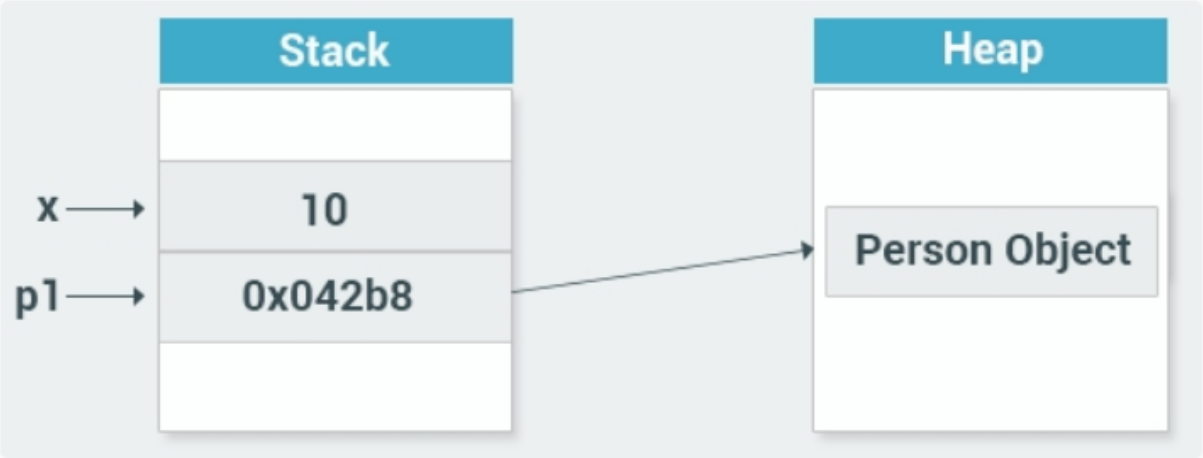
Reference Types

Reference types are used for storing objects. For example, when you create an object of a class, it is stored as a reference type.

Reference types are stored in a part of the memory called the **heap**.

When you instantiate an object, the data for that object is stored on the heap, while its heap memory address is stored on the stack.

That is why it is called a reference type - it contains a reference (the memory address) to the actual object on the heap.



As you can see, the **p1** object of type Person on the stack stores the memory address of the heap where the actual object is stored.

Stack *is used for static memory allocation, which includes all your value types, like x.*

Heap *is used for dynamic memory allocation, which includes custom objects, that might need additional memory during the runtime of your program.*

Which memory is used for dynamic allocation?

- Heap
- Stack

3. Class Example

Example of a Class

Let's create a **Person** class:

```
class Person
{
  int age;
  string name;
  public void SayHi()
  {
    Console.WriteLine("Hi");
```

```
    }
}
```

The code above declares a class named **Person**, which has **age** and **name** fields as well as a **SayHi** method that displays a greeting to the screen.

You can include an access modifier for fields and methods (also called **members**) of a class. **Access modifiers** are keywords used to specify the accessibility of a member.

A member that has been defined **public** can be accessed from outside the class, as long as it's anywhere within the scope of the class object. That is why our **SayHi** method is declared **public**, as we are going to call it from outside of the class.

*You can also designate class members as **private** or **protected**. This will be discussed in greater detail later in the course. If no access modifier is defined, the member is **private** by default.*

Fill in the blanks to create a class called Car.

```
class Car
{
  string color;
  int year;
}
```

Now that we have our Person class defined, we can instantiate an object of that type in Main.

The **new** operator instantiates an object and returns a reference to its location:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SoloLearn
{
    class Program
    {
        class Person {
            int age;
            string name;
            public void SayHi() {
                Console.WriteLine("Hi");
            }
        }
        static void Main(string[] args)
        {
            Person p1 = new Person();
            p1.SayHi();
        }
    }
}
```

The code above declares a Person object named **p1** and then calls its public **SayHi()** method.

*Notice the **dot operator (.)** that is used to access and call the method of the object.*

Fill in the blanks to create an object of type Car and call its horn() method.

```
Car c = [ new ] Car();

[ c.horn ] ();
```

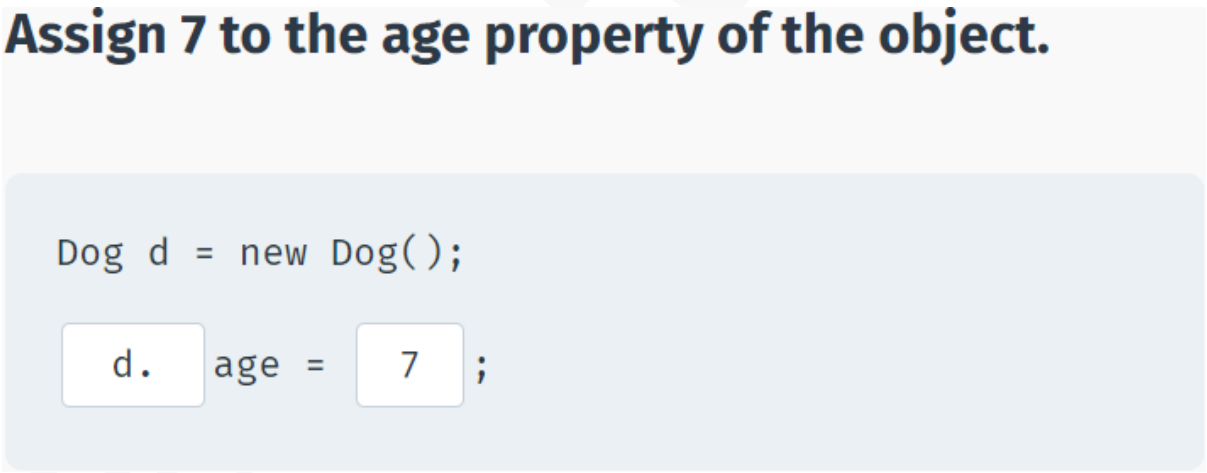
You can access all public members of a class using the dot operator.

Besides calling a method, you can use the dot operator to make an assignment when valid.

For example:

```
class Dog
{
    public string name;
    public int age;
}
static void Main(string[] args)
{
    Dog bob = new Dog();
    bob.name = "Bobby";
    bob.age = 3;

    Console.WriteLine(bob.age);
}
```



4. Encapsulation

Encapsulation

Part of the meaning of the word **encapsulation** is the idea of "surrounding" an entity, not just to keep what's inside together, but also to protect it.

In programming, encapsulation means more than simply combining members together within a class; it also means restricting access to the inner workings of that class.

Encapsulation is implemented by using **access modifiers**. An access modifier defines the scope and visibility of a class member.

*Encapsulation is also called **information hiding**.*

Encapsulation allows you to:

- Hide details of a class realization
- Assign values to variables
- Declare a method

C# supports the following access modifiers: **public, private, protected, internal, protected internal**.

As seen in the previous examples, the **public** access modifier makes the member accessible from the outside of the class.

The **private** access modifier makes members accessible only from within the class and hides them from the outside.

***protected** will be discussed later in the course.*

Which one is NOT an access modifier in C#?

- closed
- private
- internal
- protected

To show encapsulation in action, let's consider the following example:

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

namespace SoloLearn
{
    class BankAccount {
        private double balance=0;
        public void Deposit(double n) {
            balance += n;
        }
        public void Withdraw(double n) {
            balance -= n;
        }
        public double GetBalance() {
            return balance;
        }
    }
    class Program
```

```
    {
        static void Main(string[] args)
        {
            BankAccount b = new BankAccount();
            b.Deposit(199);
            b.Withdraw(42);
            Console.WriteLine(b.GetBalance());
        }
    }
}
```

We used encapsulation to hide the **balance** member from the outside code. Then we provided restricted access to it using public methods. The class data can be read through the **GetBalance** method and modified only through the **Deposit** and **Withdraw** methods.

You cannot directly change the **balance** variable. You can only view its value using the public method. This helps maintain data integrity.

We could add different verification and checking mechanisms to the methods to provide additional security and prevent errors.

In summary, the benefits of encapsulation are:

- Control the way data is accessed or modified.

- Code is more flexible and easy to change with new requirements.

- Change one part of code without affecting other parts of code.

Fill in the blanks to declare a Person class, hide the age member, and make it accessible through the GetAge method.

```
class Person {
  private int age;
  public int GetAge() {
    return age;
  }
  public void SetAge(int n) {
    age = n;
  }
}
```

5. Constructors

Constructors

A class constructor is a special member method of a class that is executed whenever a new object of that class is created.

A constructor has exactly the same name as its class, is public, and does not have any return type.

For example:

```
class Person
```

```
{
  private int age;
  public Person()
  {
    Console.WriteLine("Hi there");
  }
}
```

Now, upon the creation of an object of type Person, the constructor is automatically called.

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

namespace SoloLearn
{
    class Program
    {
        class Person
        {
            private int age;
            public Person()
            {
                Console.WriteLine("Hi there");
            }
        }
        static void Main(string[] args)
        {
            Person p = new Person();
        }
    }
}
```

This can be useful in a number of situations. For example, when creating an object of type BankAccount, you could send an email notification to the owner.

The same functionality could be achieved using a separate public method. The advantage of the constructor is that it is called automatically.

When is the constructor called?

- When a class object is created
- When the class is being declared
- Never

Constructors can be very useful for setting initial values for certain member variables.

A default constructor has no parameters. However, when needed, parameters can be added to a constructor. This makes it possible to assign an initial value to an object when it's created, as shown in the following example:

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

namespace SoloLearn
{
    class Program
    {
        class Person
        {
            private int age;
            private string name;
            public Person(string nm)
            {
                name = nm;
            }
            public string getName()
            {
```

```
                return name;
            }
        }
        static void Main(string[] args)
        {
            Person p = new Person("David");
            Console.WriteLine(p.getName());
        }
    }
}
```

Now, when the object is created, we can pass a parameter that will be assigned to the **name** variable.

*Constructors can be **overloaded** like any method by using different numbers of parameters.*

What is the output of this code?

```
class Dog
{
  public Dog()
  { Console.WriteLine(1); }

  public Dog(string name)
  { Console.WriteLine(name); }
}
static void Main(string[] args)
{
  Dog d = new Dog("2");
}
```

2

6. Properties

Properties

As we have seen in the previous lessons, it is a good practice to encapsulate members of a class and provide access to them only through public methods.

A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they actually include special methods called accessors.

The **accessor** of a property contains the executable statements that help in getting (reading or computing) or setting (writing) a corresponding field. Accessor declarations can include a **get** accessor, a **set** accessor, or both.

For example:

```
class Person
{
  private string name; //field

  public string Name //property
  {
    get { return name; }
    set { name = value; }
  }
}
```

The Person class has a **Name** property that has both the **set** and the **get** accessors.

The set accessor is used to assign a value to the name variable; get is used to return its value.

value *is a special keyword, which represents the value we assign to a property using the* ***set*** *accessor.*

The name of the property can be anything you want, but coding conventions dictate properties have the same name as the private field with a capital letter.

Which of the following are accessors?

- [x] set
- [x] get

Once the property is defined, we can use it to assign and read the private member:

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

```
namespace SoloLearn
{
    class Program
    {
        class Person
        {
            private string name;
            public string Name
            {
                get { return name; }
                set { name = value; }
            }
        }
        static void Main(string[] args)
        {
            Person p = new Person();
            p.Name = "Bob";
            Console.WriteLine(p.Name);
        }
    }
}
```

The property is accessed by its name, just like any other public member of the class.

Fill in the blanks to define valid get and set accessors for the age member:

```
class Dog
{
  private int age;
  public int Age
  {
    get { return age; }
    set { age = value; }
  }
}
```

Any accessor of a property can be omitted.

For example, the following code creates a property that is read-only:

```
class Person
{
  private string name;
  public string Name
  {
    get { return name; }
  }
}
```

*A property can also be **private**, so it can be called only from within the class.*

Skipping which accessor creates a read-only property?

- get
- set
- return

So, why use properties? Why not just declare the member variable public and access it directly?

With properties you have the option to control the logic of accessing the variable.

For example, you can check if the value of **age** is greater than 0, before assigning it to the variable:

```
class Person
{
  private int age=0;
  public int Age
  {
    get { return age; }
    set {
      if (value > 0)
        age = value;
    }
  }
}
```

*You can have any custom logic with **get** and **set** accessors.*

Fill in the blanks to create a read-only property X. The return value of the accessor should be the square of x.

```
class A {
  private int x=8;
  public int X {
     get  { return x * x; }
  }
}
```

Auto-Implemented Properties

When you do not need any custom logic, C# provides a fast and effective mechanism for declaring private members through their properties.

For example, to create a private member that can only be accessed through the **Name** property's **get** and **set** accessors, use the following syntax:

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

As you can see, you do not need to declare the private field name separately - it is created by the property automatically. **Name** is called an **auto-implemented property**. Also called auto-properties, they allow for easy and short declaration of private members.

We can rewrite the code from our previous example using an auto-property:

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

```
namespace SoloLearn
{
    class Program
    {
        class Person
        {
            public string Name { get; set; }
        }
        static void Main(string[] args)
        {
            Person p = new Person();
            p.Name = "Bob";
            Console.WriteLine(p.Name);
        }
    }
}
```

Fill in the blanks to create an auto-property named Age of type int.

```
class Dog
{
  public  int  Age {  get;  set; }
}
```

7. Module 4 Quiz

Objects of a class are stored on the:

- heap
- property
- stack

Fill in the blanks to instantiate an object of the class Cat, passing to the constructor the value 12. Then call the Meow method for that object:

```
Cat c = new Cat( 12 );

c. Meow();
```

To make a member of a class accessible from outside the class declaration, you should declare it as:

- public
- void
- private

Which statement is true?

- An object is a method of a class.
- An object is an instance of a class.
- An object is a member of a class.

Fill in the blanks to declare a class Student, with one public method called Hello. The Hello method displays "hi" to the screen.

```
class Student {
  public void Hello() {
    Console.WriteLine("hi");
  }
}
```

Fill in the blanks to declare a constructor that has one parameter and assigns it to the age member:

```
class Dog
{
  private int age;
    public  Dog( int  val)
  {
    age =  val ;
  }
}
```

Which accessor is used to read the value of a member?

- void
- set
- get
- return

5. Arrays & Strings

1. Arrays

Arrays

C# provides numerous built-in classes to store and manipulate data.

One example of such a class is the **Array** class.

An array is a data structure that is used to store a collection of data. You can think of it as a collection of variables of the **same type**.

For example, consider a situation where you need to store 100 numbers. Rather than declare 100 different variables, you can just declare an array that stores 100 **elements**.

To declare an array, specify its element types with square brackets:

```
int[] myArray;
```

This statement declares an array of integers.

Since arrays are objects, we need to instantiate them with the **new** keyword:

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

This instantiates an array named myArray that holds 5 integers.

*Note the **square brackets** used to define the number of elements the array should hold.*

Fill in the blanks to instantiate an array of 42 elements of type double:

```
double [ ] a = new double[ 42 ];
```

After creating the array, you can assign values to individual elements by using the **index number**:

```
int[] myArray = new int[5];
myArray[0] = 23;
```

This will assign the value 23 to the first element of the array.

Arrays in C# are zero-indexed meaning the first member has index 0, the second has index 1, and so on.

The third element of an array has index:

2

We can provide initial values to the array when it is declared by using curly brackets:

```
string[] names = new string[3] {"John", "Mary", "Jessica"};
double[] prices = new double[4] {3.6, 9.8, 6.4, 5.9};
```

We can omit the size declaration when the number of elements are provided in the curly braces:

```
string[] names = new string[] {"John", "Mary", "Jessica"};
double[] prices = new double[] {3.6, 9.8, 6.4, 5.9};
```

We can even omit the **new** operator. The following statements are identical to the ones above:

```
string[] names = {"John", "Mary", "Jessica"};
double[] prices = {3.6, 9.8, 6.4, 5.9};
```

Array values should be provided in a comma separated list enclosed in {curly braces}.

Fill in the blanks to instantiate the array with initial values:

```
int[ ] a = { 1, 2, 3 } ;
```

As mentioned, each element of an array has an index number.

For example, consider the following array:

```
int[] b = {11, 45, 62, 70, 88};
```

The elements of **b** have the following indexes:

11	45	62	70	88
[0]	[1]	[2]	[3]	[4]

To access individual array elements, place the element's index number in square brackets following the array name.

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] b = {11, 45, 62, 70, 88};

            Console.WriteLine(b[2]);
            Console.WriteLine(b[3]);
        }
    }
}
```

Remember that the first element has index 0.

What is the value of x after these statements execute?

```
int[ ] a = {4, 7, 2};
int x = a[0]+a[2];
```

6

2. Using Arrays in Loops

Arrays & Loops

It's occasionally necessary to iterate through the elements of an array, making element assignments based on certain calculations. This can be easily done using loops.

For example, you can declare an array of 10 integers and assign each element an even value with the following loop:

```
int[] a = new int[10];
for (int k = 0; k < 10; k++) {
  a[k] = k*2;
}
```

We can also use a loop to read the values of an array.

For example, we can display the contents of the array we just created:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            int[] a = new int[10];
            for (int k = 0; k < 10; k++) {
                a[k] = k*2;
            }
            for (int k = 0; k < 10; k++) {
                Console.WriteLine(a[k]);
            }
        }
    }
}
```

This will display the values of the elements of the array.

*The variable **k** is used to access each array element.*

The last index in the array is 9, so the for loop condition is k<10.

Fill in the blanks to print all elements of the array using a for loop.

```
int[ ] arr = new int[7];

   for   (int k=0;k<   7   ;k++) {

  Console.WriteLine(   arr   [k]);

}
```

The foreach Loop

The **foreach loop** provides a shorter and easier way of accessing array elements.

The previous example of accessing the elements could be written using a **foreach loop**:

```
using System;
```

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int[ ] a = new int[10];
            for (int k = 0; k < 10; k++) {
                a[k] = k*2;
            }
            foreach (int k in a) {
                Console.WriteLine(k);
            }
        }
    }
}
```

The **foreach loop** iterates through the array a and assigns the value of the current element to the variable **k** at each iteration of the loop. So, at the first iteration, k=a[0], at the second, k=a[1], etc.

*The data type of the variable in the **foreach loop** should match the type of the array elements.*

*Often the keyword **var** is used as the type of the variable, as in: **foreach (var k in a)**. The compiler determines the appropriate type for var.*

Fill in the blanks to create a valid foreach loop that displays all even elements of the array.

```
int[ ] nums = {5, 2, 3, 4, 7};

    foreach    (var n   in   nums) {

  if(n%2 == 0)

    Console.WriteLine(   n   );

}
```

Arrays

The following code uses a **foreach** loop to calculate the sum of all the elements of an array:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int[ ] arr = {11, 35, 62, 555, 989};
            int sum = 0;

            foreach (int x in arr) {
```

```
                    sum += x;
                }
                Console.WriteLine(sum);
            }
        }
}
```

To review, we declared an array and a variable **sum** that will hold the sum of the elements.

Next, we utilized a **foreach** loop to iterate through each element of the array, adding the corresponding element's value to the **sum** variable.

*The **Array** class provides some useful methods that will be discussed in the coming lessons.*

What is the output of this code?

```
int[ ] arr = {8, 2, 6};
int y=0;
foreach (int x in arr) {
  y+=x/2;
}
Console.Write(y);
```

8

3. Multidimensional Arrays

Multidimensional Arrays

An array can have multiple dimensions. A **multidimensional array** is declared as follows:

```
type[, , … ,] arrayName = new type[size1, size2, …, sizeN];
```

For example, let's define a two-dimensional 3x4 integer array:

```
int[ , ] x = new int[3,4];
```

Visualize this array as a table composed of 3 rows and 4 columns:

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Array indexing starts from 0.

Fill in the blanks to declare a two-dimensional array of integers with 8 rows and 8 columns.

```
int[ , ] chessBoard = new int[8, 8 ];
```

We can initialize multidimensional arrays in the same way as single-dimensional arrays.

For example:

```
int[ , ] someNums = { {2, 3}, {5, 6}, {4, 6} };
```

This will create an array with three rows and two columns. Nested curly brackets are used to define values for each row.

To access an element of the array, provide both indexes. For example **someNums[2, 0]** will return the value **4**, as it accesses the first column of the third row.

Let's create a program that will display the values of the array in the form of a table.

```
using System;
```

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int[ , ] someNums = { {2, 3}, {5, 6}, {4, 6} };
            for (int k = 0; k < 3; k++) {
                for (int j = 0; j < 2; j++) {
                    Console.Write(someNums[k, j]+" ");
                }
                Console.WriteLine();
            }
        }
    }
}
```

We have used two nested **for** loops, one to iterate through the rows and one through the columns.

The Console.WriteLine(); statement moves the output to a new line after one row is printed.

Arrays can have any number of dimensions, but keep in mind that arrays with more than three dimensions are harder to manage.

How many dimensions does the following array have?

```
int[ , , , ] arr;
```

- 3
- 5
- 4

4. Jagged Arrays

Jagged Arrays

A **jagged array** is an array whose elements are arrays. So it is basically an **array of arrays**.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[ ][ ] jaggedArr = new int[3][ ];
```

Each dimension is an array, so you can also initialize the array upon declaration like this:

```
int[ ][ ] jaggedArr = new int[ ][ ]
{
  new int[ ] {1,8,2,7,9},
  new int[ ] {2,4,6},
  new int[ ] {33,42}
};
```

You can access individual array elements as shown in the example below:

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

```
using System.Threading.Tasks;

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int[ ][ ] jaggedArr = new int[ ][ ]
            {
                new int[ ] {1,8,2,7,9},
                new int[ ] {2,4,6},
                new int[ ] {33,42}
            };
            int x = jaggedArr[2][1];
            Console.WriteLine(x);
        }
    }
}
```

This accesses the second element of the third array.

*A **jagged array** is an array-of-arrays, so an int[ ][ ] is an array of int[ ], each of which can be of different lengths and occupy their own block in memory.*

*A **multidimensional array** (int[,]) is a single block of memory (essentially a matrix). It always has the same amount of columns for every row.*

Fill in the blanks to declare a jagged array that contains 8 two-dimensional arrays.

```
int[ ][,] a = new int[  8  ][  ,  ];
```

5. Array Properties & Methods

Arrays Properties

The Array class in C# provides various properties and methods to work with arrays.

For example, the **Length** and **Rank** properties return the number of elements and the number of dimensions of the array, respectively. You can access them using the dot syntax, just like any class members:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            int[ ] arr = {2, 4, 7};

            Console.WriteLine(arr.Length);

            Console.WriteLine(arr.Rank);
        }
    }
}
```

The **Length** property can be useful in **for** loops where you need to specify the number of times the loop should run.

For example:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            int[ ] arr = {2, 4, 7};
            for(int k=0; k<arr.Length; k++) {
                Console.WriteLine(arr[k]);
            }
        }
    }
}
```

What is the output of this code?

```
int[ , , ] a = new int[2, 3, 4];
Console.Write(a.Rank);
```

3

Array Methods

There are a number of methods available for arrays.

Max returns the largest value.

Min returns the smallest value.

Sum returns the sum of all elements.

For example:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            int[ ] arr = { 2, 4, 7, 1};
            Console.WriteLine(arr.Max());
            Console.WriteLine(arr.Min());
            Console.WriteLine(arr.Sum());
        }
    }
}
```

C# also provides a static Array class with additional methods. You will learn about those in the next module.

What is the output of this code?

```
int[ ] a = {4, 6, 5, 2};
int x = a[0]+a.Min();
Console.Write(x);
```

6

6. Working with Strings

Strings

It's common to think of strings as arrays of characters. In reality, strings in C# are objects.

When you declare a **string** variable, you basically instantiate an object of type **String**.

String objects support a number of useful properties and methods:

Length returns the length of the string.

IndexOf(value) returns the index of the first occurrence of the value within the string.

Insert(index, value) inserts the value into the string starting from the specified index.

Remove(index) removes all characters in the string from the specified index.

Replace(oldValue, newValue) replaces the specified value in the string.

Substring(index, length) returns a substring of the specified length, starting from the specified index. If length is not specified, the operation continues to the end of the string.

Contains(value) returns true if the string contains the specified value.

The examples below demonstrate each of the String members:

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

namespace SoloLearn
{
    class Program
    {
        static void Main(string[] args)
        {
            string a = "some text";
            Console.WriteLine(a.Length);
            //Outputs 9

            Console.WriteLine(a.IndexOf('t'));
            //Outputs 5

             a = a.Insert(0, "This is ");
            Console.WriteLine(a);
            //Outputs "This is some text"

            a = a.Replace("This is", "I am");
            Console.WriteLine(a);
            //Outputs "I am some text"

            if(a.Contains("some"))
                Console.WriteLine("found");
            //Outputs "found"

            a = a.Remove(4);
            Console.WriteLine(a);
            //Outputs "I am"
```

```
            a = a.Substring(2);
            Console.WriteLine(a);
            //Outputs "am"
        }
    }
}
```

You can also access characters of a string by its index, just like accessing elements of an array:

```
static void Main(string[] args)
{
    string a = "some text";
    Console.WriteLine(a[2]);
}
```

Indexes in strings are similar to arrays, they start from 0.

What is the output of this code?

```
string s = "SoloLearn is awesome";
Console.Write(s.IndexOf('e'));
```

5

Working with Strings

Let's create a program that will take a string, replace all occurrences of the word "dog" with "cat" and output the first sentence only.

```
static void Main(string[] args)
{
    string text = "This is some text about a dog. The word dog
appears in this text a number of times. This is the end.";
    text = text.Replace("dog", "cat");
    text = text.Substring(0, text.IndexOf(".")+1);

    Console.WriteLine(text);
}
```

The code above replaces all occurrences of "dog" with "cat". After that it takes a substring of the original string starting from the first index until the first occurrence of a period character.

We add one to the index of the period to include the period in the substring.

C# provides a solid collection of tools and methods to work and manipulate strings. You could, for example, find the number of times a specific word appears in a book with ease, using those methods.

Fill in the blanks to assign the last character of the string to the x variable:

```
string s = "Hello";

char x;

x = [ s ][s.Length-1[ ]];
```

7. Module 5 Quiz

An array is a:

- reference type
- none of these
- value type

Fill in the blanks to print all elements of the array.

```
int[] arr = {0, 5, 3, 2, 1 };

foreach (int item  in  arr) {

  Console.WriteLine(  item  );

}
```

What is the output of this code?
string s = "SoloLearn";
Console.Write(s[6]);

a

How many elements can the following array store?

```
int[ , , ] array = new int[4, 5, 3];
```

- 12
- 4
- 60
- 59

What is the output of this code?

```
string s = "SoloLearn";
int x = s.Length;
int y = s.IndexOf("e");
Console.Write(x%y);
```

- 4

6. More On Classes

1. Destructors

7. Inheritance & Polymorphism

8. Structs, Enums, Exceptions & Files

9. Generics

Dictionary

Noun

basic concepts : những khái niệm cơ bản

language : ngôn ngữ

applications : những ứng dụng

Web services : dịch vụ web

client-server : máy khách-máy chủ

database : cơ sở dữ liệu

.NET Framework : là một nền tảng lập trình

Common Language Runtime : ngôn ngữ thực thi tổng quát

class library : lớp thư viện

foundation : nền tảng

memory management : quản lý bộ nhớ

collection : bộ sưu tập

aspects of code : các khía cạnh của code

execution time : thời gian thực hiện

accuracy : sự chính xác

core services : những dịch vụ cốt lõi

task : nhiệm vụ

various functionalities : các chức năng khác nhau

components : các thành phần

variables : biến (ví dụ $f(x) = x^2$ ->x chính là biến trong hàm)

memory location : vị trí bộ nhớ

name : tên

data type : kiểu dữ liệu

underscore character : '_' dấu gạch dưới

semicolon : ';' dấu chấm phẩy

commas : ',' dấu phẩy

single quotes : " ' " dấu nháy đơn

parentheses : '(' dấu ngoặc đơn

curly braces : '{' dấu ngoặc nhọn

quotation marks : ' " ' dấu ngoặc kép

two slashes : '//' 2 dấu xẹt
information : thông tin
identifier : sự định danh
statement : câu lệnh
operation : sự điều hành
sequence : sự liên tục
character : chữ cái
additional software : phần mềm bổ sung
classes : các lớp
methods : các phương thức
arguments : các đối số
namespaces : không gian tên
general message : thông báo chung
text-only interface : giao diện thuần văn bản
a line terminator : dấu xuống dòng
console : bàn điều khiển
cursor : con trỏ
formatted string : chuỗi định dạng
syntax : cú pháp
conversion : sự chuyển đổi
alternatives : lựa chọn thay thế
default : mặc định
precedence : quyền ưu tiên
Operator : hệ điều hành
compound assignment operators : toán tử gán ghép
Prefix : tiền tố
Postfix : hậu tố
nested if Statements : lệnh if lồng nhau
equality : sự bằng nhau
increment : sự gia tăng
decrement : sự giảm dần
iteration : sự lặp lại

loop : vòng lặp
expression : biểu thức
reusable code : tái sử dụng code
parameter : tham số
rectangle : hình chữ nhật
Recursion : đệ quy
Pyramid : kim tự tháp
 algorithm : thuật toán
properties : tính chất
instance : ví dụ
instantiation : sự tức thời
characteristic : nét đặc trưng
Stack : ngăn xếp - bộ nhớ tĩnh lưu các biến biết trước dung lượng
Heap : chất đống - bộ nhớ động lưu các dữ liệu ko biết trước dung lượng
encapsulation : sự đóng gói
access modifier : công cụ sửa đổi truy cập
constructors : hàm khởi tạo
convention : quy ước
accessors : người truy cập
Arrays : mảng
Strings : chuỗi
elements : phần tử
situation : tình huống
index : chỉ mục
Jagged Arrays : mảng răng cưa

Verb

enable : cho phép, kích hoạt

build : xây dựng

create : chế tạo

run : chạy

consists of : bao gồm

collect : sưu tầm

accomplish : đạt được

file access : truy cập file

reserves : dự trữ

store : lưu trữ

declare : khai báo

specify : xác định

complete : hoàn thành

descriptive of : mô tả về

contain : lưu trữ

perform : biểu diễn

install : cài đặt

Pressing any key : nhấn phím bất kỳ

pass data : truyền dữ liệu

display : trưng bày

include : bao gồm

prompt : nhắc nhở

assign : chỉ định

convert : chuyển đổi

expect : mong đợi

follow : theo

facilitate : tạo điều kiện

alter : thay đổi

evaluate : đánh giá

continue : tiếp tục

break : phá vỡ

indicate : biểu thị

define : định nghĩa

input : đầu vào

output : đầu ra

value : theo giá trị

reference : theo tham chiếu

concatenate values : nối các giá trị

determine : xác định

represent : đại diện

omit : bỏ sót

Adjective

elegant : thanh lịch

variety of : đa dạng

secure : an toàn

robust : mạnh mẽ

upcoming : sắp tới

complex : phức tạp

important : quan trọng

understandable : có thể hiểu được

readable : có thể đọc được

True : đúng

False : sai

different : khác nhau

empty : bỏ trống

explanatory : được giải thích

conditional : có điều kiện

entire : toàn bộ

distinguished : được phân biệt

dynamic : năng động

private : riêng tư

public : cộng đồng

protected : được bảo vệ

flexible : linh hoạt

identical : giống hệt

Other

such as : như là

extensively : một cách chuyên sâu

automatically : một cách tự động

implicitly : ngầm hiểu

explicitly : rõ ràng

indefinitely : vô thời hạn

foreach : cho mỗi

End