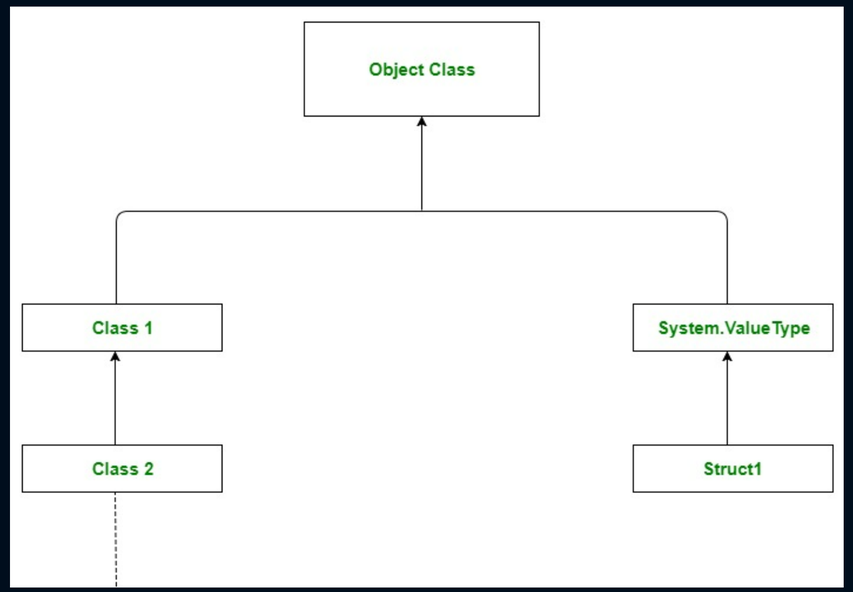
System.Object

The Object class is the base class for all the classes in the .Net Framework. It is present in the System namespace. In C#, the .NET Base Class Library(BCL) has a language-specific alias which is Object class with the fully qualified name as System.Object.
Every class in C# is directly or indirectly derived from the Object class, this is implicit.
The main purpose of the Object class is to provide low-level services to derived classes.



C# object class

Inheritance from Object

Methods	Description
Equals(Object)	Determines whether the specified object is equal to the current object.
Equals(Object, Object)	Determines whether the specified object instances are considered equal.
Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
GetHashCode()	Serves as the default hash function.
GetType()	Gets the Type of the current instance.
MemberwiseClone()	Creates a shallow copy of the current Object.
ReferenceEquals(Object, Object)	Determines whether the specified Object instances are the same instance.
ToString()	Returns a string that represents the current object.

System.Object Methods

Object methods

Primitive types

C# type/keyword	Range	Size	.NET type
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
`nint`	Depends on platform (computed at runtime)	Signed 32-bit or 64-bit integer	System.IntPtr
`nuint`	Depends on platform (computed at runtime)	Unsigned 32-bit or 64-bit integer	System.UIntPtr

`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes	System.Single
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes	System.Double
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes	System.Decimal

Data types

Using Primitive types

Using Primitive types

```
int population = 66_000_000; // 66 million in UK
double weight = 1.88; // in kilograms
decimal price = 4.99M; // in pounds sterling
string fruit = "Apples"; // strings use double-quotes
char letter = 'Z'; // chars use single-quotes
bool happy = true; // Booleans have value of true or
false
```

Primitive types and keywords

Keyword	Aliased type
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Using strings

String interpolation and scape special characters

Given:

```
string text = "red";
int number = 14;
const int width = -4;
```

Then:

Interpolated String Expression	Equivalent Meaning As `string`	Value
`$"{text}"`	`string.Format("{0}", text)`	`"red"`
`$"{{text}}"`	`string.Format("{{text}})`	`"{text}"`
`$"{ text , 4 }"`	`string.Format("{0,4}", text)`	`" red"`
`$"{ text , width }"`	`string.Format("{0,-4}", text)`	`"red "`
`$"{number:X}"`	`string.Format("{0:X}", number)`	`"E"`
`$"{text + '?'} {number % 3}"`	`string.Format("{0} {1}", text + '?', number % 3)`	`"red? 2"`
`$"{text + $"[{number}]"}"`	`string.Format("{0}", text + string.Format("[{0}]", number))`	`"red[14]"`
`$"{(number==0?"Zero":"Non-zero")}"`	`string.Format("{0}", (number==0?"Zero":"Non-zero"))`	`"Non-zero"`

end example

Expression

Same expresión using Format String

Result

Declaring string type variables

```
string firstName = "Bob"; // assigning literal strings
string lastName = "Smith";
string phoneNumber = "(215) 555-4256";
// assigning a string returned from a fictitious function
string address = GetAddressFromDatabase(id: 563);
```

Formatting

```
Console.WriteLine($"{numberOfApples} apples costs {pricePerApple * numberOfApples:C}");
```

```
12 apples costs £4.20
```

→ Resultado

The (:C) value in numberOfApples:C applies a currency format

Enumerations

Data type that groups named constants, it is used for improve readability.
It is a Value type and it uses the enum Keyword.
Enums provide a way to represent a finite set of distinct values as a single type.

Enums are commonly used to represent: State or status, options, flags and bitwise operations.

Enumeration example

```
enum LocationTypes
{
    Campus, //0
    Building, //1
    Floor, //2
    Room //3
}
```

Date time and Timespan applications

Date Time is a data type in C# used to represent dates and times. It provides properties such as Year, Month, Day, Hour, Minute, Second, and Millisecond.
It supports formatting and parsing of dates and times using standard or custom format strings. Can be compared, added, subtracted, and manipulated using arithmetic and conversion methods.
Time Span is a data type in C# used to represent a duration or elapsed time. It is useful for performing calculations involving time durations or intervals.

Date Time

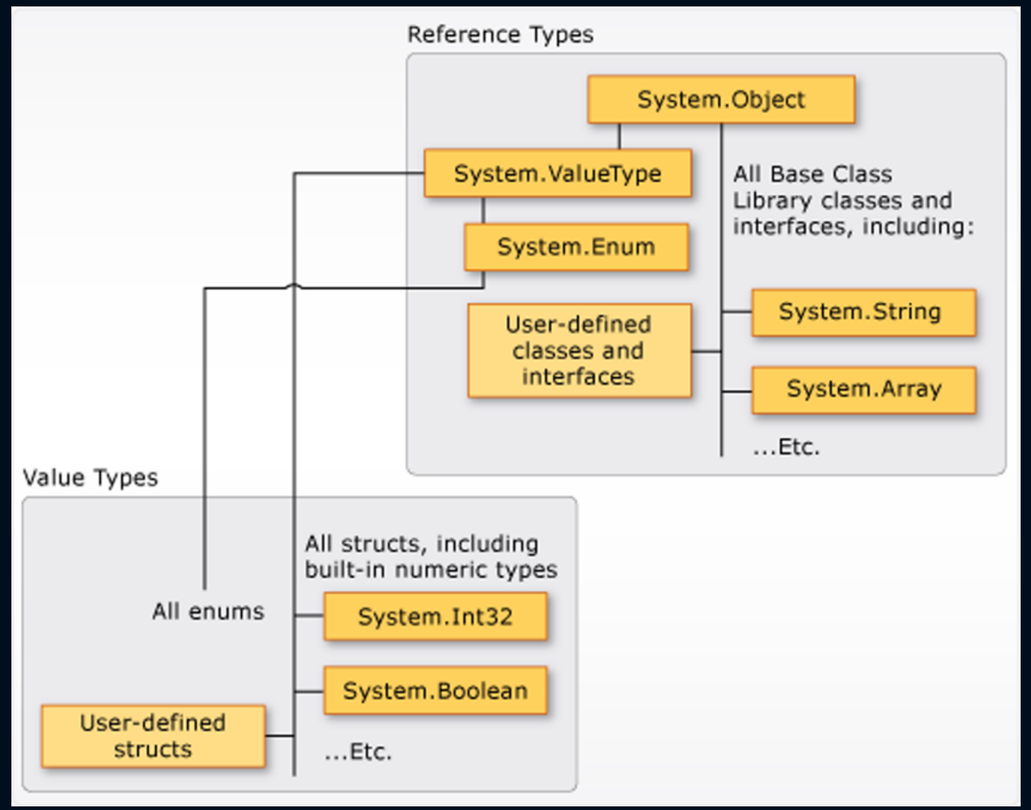
```
public static void UsingDateTime()
{
  var now = DateTime.Now;
  var customDate = new DateTime(2023, 06, 05);
  var tommorrowDate = now.AddDays(1);
}
```

Time Span

```
public static void UsingTimeSpan()
{
    var timeBetween = DateTime.Now.Subtract(DateTime.Now.AddDays(1));
    var time = new TimeSpan(12, 30, 40);
}
```

Common Type System

The Common Type System (CTS) is a set of rules and definitions that describe the data types that can be used in C#. The CTS defines primitive data types, structures, classes, enumerations, reference types, and value types.



Common Type System

Some uncommon operators

Bitwise

```
int a = 10; // 00001010
int b = 6;  // 00000110
WriteLine($"a = {a}");
WriteLine($"b = {b}");
WriteLine($"a & b = {a & b}"); // 2-bit column only
WriteLine($"a | b = {a | b}"); // 8, 4, and 2-bit
columns
WriteLine($"a ^ b = {a ^ b}"); // 8 and 4-bit columns
```

Binary Shift

```
// 01010000 left-shift a by three bit columns
WriteLine($"a << 3 = {a << 3}");
// multiply a by 8
WriteLine($"a * 8 = {a * 8}");
// 00000011 right-shift b by one bit column
WriteLine($"b >> 1 = {b >> 1}");
```

Some uncommon operators

Miscellaneous

The following operations in C# are subject to binding:

- Member access: `e.M`
- Method invocation: `e.M(e₁,...,eᵥ)`
- Delegate invocation: `e(e₁,...,eᵥ)`
- Element access: `e[e₁,...,eᵥ]`
- Object creation: new `C(e₁,...,eᵥ)`
- Overloaded unary operators: `+, -, !, ~, ++, --, true, false`
- Overloaded binary operators: `+, -, *, /, %, &, &&, |, ||, ??, ^, <<, >>, ==, !=, >, <, >=, <=`
- Assignment operators: `=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`
- Implicit and explicit conversions

Logical

```
bool a = true;
bool b = false;
WriteLine($"AND   | a     | b     ");
WriteLine($"a     | {a & a,-5} | {a & b,-5} ");
WriteLine($"b     | {b & a,-5} | {b & b,-5} ");
WriteLine();
WriteLine($"OR    | a     | b     ");
WriteLine($"a     | {a | a,-5} | {a | b,-5} ");
WriteLine($"b     | {b | a,-5} | {b | b,-5} ");
WriteLine();
WriteLine($"XOR   | a     | b     ");
WriteLine($"a     | {a ^ a,-5} | {a ^ b,-5} ");
WriteLine($"b     | {b ^ a,-5} | {b ^ b,-5} ");
```

Operators

Level	Category	Operators	Associativity (binary/tertiary)
14	Primary	`x.y  f(x)  a[x]  x++  x--` `new  typeof  checked  unchecked  default  delegate`	left to right
13	Unary	`+  -  !  ~  ++x  --x  (T)x  true  false  sizeof`	left to right
12	Multiplicative	`*  /  %`	left to right
11	Additive	`+  -`	left to right
10	Shift	`<<  >>`	left to right
9	Relational and Type testing	`<  <=  >  >=  is  as`	left to right
8	Equality	`==  !=`	left to right
7	Logical/bitwise and	`&`	left to right
6	Logical/bitwise xor	`^`	left to right
5	Logical/bitwise or	`|`	left to right
4	Conditional and	`&&`	left to right
3	Conditional or	`||`	left to right
2	Null coalescing	`??`	left to right
1	Conditional	`?:`	right to left
0	Assignment or Lambda expression	`=  *=  /=  %=  +=  -=` `<<=  >>=  &=  ^=  |=  =>`	right to left

Conditionals

If-Else Statements: Used to perform different actions based on a condition. It can be nested or combined with else-if (else if) statements to handle multiple conditions.

Switch Statements: Used to perform different actions based on the value of a variable or expression. The break statement is used to exit the switch statement after executing the corresponding case.

Ternary Operator: Provides a concise way to write conditional expressions with a single line of code.

if else

```
if (condition)
{
    // Code to execute if condition is true
}
else
{
    // Code to execute if condition is false
}
```

switch

```
switch (variable/expression)
{
    case value1:
        // Code to execute if variable/expression matches value1
        break;
    case value2:
        // Code to execute if variable/expression matches value2
        break;
    // Add more cases as needed
    default:
        // Code to execute if no case matches
        break;
}
```

ternary operator

```
condition ? expression1 : expression2
```

Loops

For: Used when the number of iterations is known or when iterating over a range or collection.

While: Used when the number of iterations is known or when iterating over a range or collection.

Do-While: Similar to While but the code block is executed at least once before checking the condition.

Foreach: Used to iterate over elements in a collection or array.

for

```
for (initialization; condition; iteration)
{
    // Code to execute in each iteration
}
```

while

```
while (condition)
{
    // Code to execute while the condition is true
}
```

do while

```
do
{
    // Code to execute
} while (condition);
```

foreach

```
foreach (var item in collection)
{
    // Code to execute for each item
}
```

Methods

Methods are a fundamental concept in C# that allow us to encapsulate a block of code and give it a name, promoting code reusability, modularity, and organization.

Syntax

Access Modifier: Determines the visibility and accessibility of the method (e.g., public, private, etc.).
Return Type: Specifies the type of value the method returns (void if no return value).
Method Name: The unique name used to call the method.
Parameters: Input values passed to the method (optional).

Method syntax

```
<accessModifier> <returnType> MethodName(Parameters)
{
    // Code to be executed
    // Optional return statement
}
```

Example: Method without return

```
public void DisplayResult(int number1, int number2)
{
    int result = a * b;
    Console.WriteLine("The result is " + result);
}
```

Example: Method that returns int

```
public int MultiplyTwoNumbers(int number1, int number2)
{
    return a * b;
}
```

Obtaining input information

ReadLine Method

```
Console.Write("Type your first name and press ENTER:
");
string? firstName = Console.ReadLine();
Console.Write("Type your age and press ENTER: ");
string? age = Console.ReadLine();
Console.WriteLine(
  $"Hello {firstName}, you look good for {age}.");
```

Displaying message

Waiting for Input

Waiting for Input

```
Type your name and press ENTER: Gary
Type your age and press ENTER: 34
Hello Gary, you look good for 34.
```

Result

ReadKey Method

```
using static System.Console;
```

Importing namespaces

```
Write("Press any key combination: ");
ConsoleKeyInfo key = ReadKey();
WriteLine();
WriteLine("Key: {0}, Char: {1}, Modifiers: {2}",
  arg0: key.Key,
  arg1: key.KeyChar,
  arg2: key.Modifiers);
```

Displaying message

Waiting for Key Input

```
Press any key combination: k
Key: K, Char: k, Modifiers: 0
```

Result

Demo

We will check the following:

Using C# syntax

Using conversions

Using operators

You can check the following Repository for some examples:

C# fundamentals

Tasks

Exercise/ Homework

Create a task manager basic console application. This will allow users to add tasks, mark tasks as completed, and view their tasks. The application should run in a loop only exiting when the user selects a specific option. Next, you can see the workflow:

```
Choose an option:
1. Add Task
2. View Tasks
3. Mark Task as Completed
4. Exit
Enter your choice (1-4):
```

```
Enter your choice (1-4): 4
Exiting the Task Manager. Goodbye!
```

```
Choose an option:
1. Add Task
2. View Tasks
3. Mark Task as Completed
4. Exit
Enter your choice (1-4): 1
Enter the task description: Task 1
Task added successfully!
```

```
Choose an option:
1. Add Task
2. View Tasks
3. Mark Task as Completed
4. Exit
Enter your choice (1-4): 2

Your Tasks:
1. Task 1
```

```
Choose an option:
1. Add Task
2. View Tasks
3. Mark Task as Completed
4. Exit
Enter your choice (1-4): 3

Your Tasks:
1. Task 1
Enter the task number to mark as completed: 1
Task marked as completed!
```

```
Choose an option:
1. Add Task
2. View Tasks
3. Mark Task as Completed
4. Exit
Enter your choice (1-4): 2

Your Tasks:
No tasks available.
```