

C# Tutorial

Based on
<http://www.tutorialsteacher.com>
<https://www.sololearn.com>
and etc.

C# is a simple & powerful object-oriented programming language developed by Microsoft and approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO).

C# was developed by Anders Hejlsberg and his team during the development of .Net Framework.

C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages on different computer platforms and architectures.

In order to use C# with your .Net application, you need two things, .NET Framework and IDE (Integrated Development Environment).

The .NET Framework:

The .NET Framework is a platform where you can write different types of web and desktop based applications. You can use C#, Visual Basic, F# and Jscript to write these applications.

Integrated Development Environment (IDE):

An IDE is a tool that helps you write your programs. Visual Studio is an IDE provided by Microsoft to write the code in languages such as C#, F#, VisualBasic, etc.

C# Class:

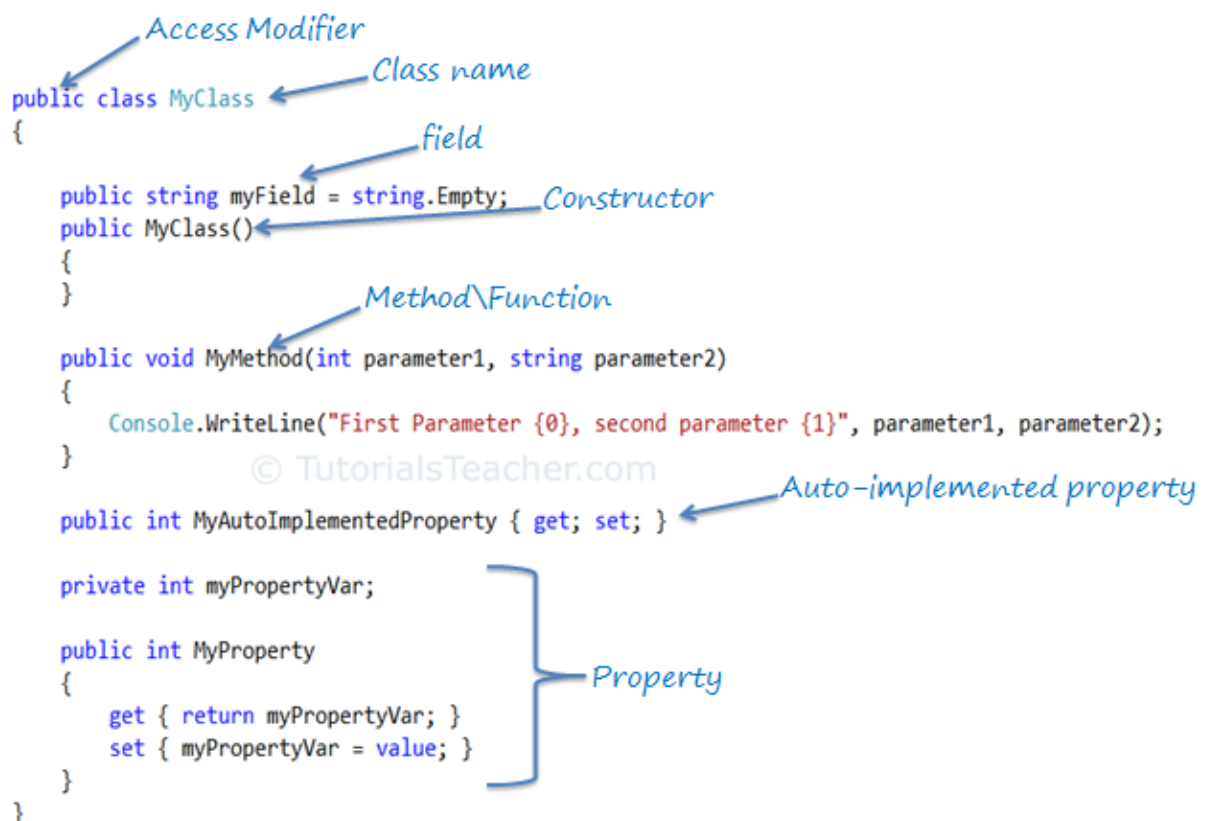
A class is like a blueprint of specific object.

It defines certain properties, fields, events, method etc. A class defines the kinds of data and the functionality their objects will have.

A class enables you to create your own custom types by grouping together variables of other types, methods and events.

In C#, a class can be defined by using the class keyword.

The following image shows the important building blocks of C# class.



The diagram illustrates the building blocks of a C# class with the following code and annotations:

```
public class MyClass
{
    public string myField = string.Empty;
    public MyClass()
    {
    }
    public void MyMethod(int parameter1, string parameter2)
    {
        Console.WriteLine("First Parameter {0}, second parameter {1}", parameter1, parameter2);
    }
    public int MyAutoImplementedProperty { get; set; }
    private int myPropertyVar;
    public int MyProperty
    {
        get { return myPropertyVar; }
        set { myPropertyVar = value; }
    }
}
```

Annotations in the diagram:

- Access Modifier**: points to `public` in `public class MyClass`.
- Class name**: points to `MyClass` in `public class MyClass`.
- field**: points to `myField` in `public string myField = string.Empty;`.
- Constructor**: points to `MyClass()` in `public MyClass()`.
- Method\Function**: points to `MyMethod` in `public void MyMethod(int parameter1, string parameter2)`.
- Auto-implemented property**: points to `MyAutoImplementedProperty` in `public int MyAutoImplementedProperty { get; set; }`.
- Property**: points to the `MyProperty` block (including `private int myPropertyVar;` and the `get/set` methods).

© TutorialsTeacher.com

Access Modifiers:

Access modifiers are applied on the declaration of the class, method, properties, fields and other members. They define the accessibility of the class and its members. Public, private, protected and internal are access modifiers in C#. We will learn about it in the [keyword](#) section.

Field:

Field is a class level variable that can hold a value. Generally field members should have a private access modifier and used with a property.

Constructor:

A class can have parameterized or parameter less constructors. The constructor will be called when you create an instance of a class. Constructors can be defined by using an access modifier and class name: `<access modifiers> <class name>(){ }`

Method:

A method can be defined using the following template:

```
{access modifier} {return type} MethodName({parameterType parameterName})
```

Property:

A property can be defined using getters and setters

Property encapsulates a private field. It provides getters (get{}) to retrieve the value of the underlying field and setters (set{}) to set the value of the underlying field. In the above example, `_myPropertyVar` is a private field which cannot be accessed directly. It will only be accessed via `MyProperty`. Thus, `MyProperty` encapsulates `_myPropertyVar`.

You can also apply some addition logic in get and set, as in the below example.

```
private int _myPropertyVar;

public int MyProperty
{
    get {
        return _myPropertyVar / 2;
    }

    set {
        if (value > 100)
            _myPropertyVar = 100;
        else
            _myPropertyVar = value;
    }
}
```

Auto-implemented Property:

From C# 3.0 onwards, property declaration has been made easy if you don't want to apply some logic in get or set.

The following is an example of an auto-implemented property:

```
public int MyAutoImplementedProperty { get; set; }
```

Notice that there is no private backing field in the above property example. The backing field will be created automatically by the compiler. You can work with an automated property as you would with a normal property of the class. Automated-implemented property is just for easy declaration of the property when no additional logic is required in the property accessors.

Namespace:

Namespace is a container for a set of related classes and namespaces. Namespace is also used to give unique names to classes within the namespace name. Namespace and classes are represented using a dot (.).

```
namespace CSharpTutorials
{
    class MyClass
    {
    }
}
```

In the above example, the fully qualified class name of MyClass is CSharpTutorials.MyClass.

A namespace can contain other namespaces. Inner namespaces can be separated using (.).

```
namespace CSharpTutorials.Examples
{
    class MyClassExample
    {
    }
}
```

In the above example, the fully qualified class name of MyClassExample is `CSharpTutorials.Example.MyClassExample`

Points to Remember :

1. **C# Class** defines properties, fields, events, methods etc. An object is a instance of the class.
2. Access modifiers defines the accessbility of a class e.g. public, private, protected or internal.
3. **Namespace** can include one or more classes.

C# Variable:

The variable in C# is nothing but a name given to a data value.

In C#, a variable is always defined with a [data type](#).

Example: Variable declaration & initialization

```
string message = "Hello World!!";  
char y = 'Z';
```

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

Multiple variables of the same data type can be declared and initialized in a single line separated by commas.

Example: Multiple variable declaration

```
int i, j, k, l = 0;  
int amount, num;
```

The var Keyword

As we already know 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.

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;
```

Var can be used in the following different contexts:

- Local variable in a function
- For loop
- Foreach loop
- Using statement
- As an anonymous type
- In a LINQ query expression

var cannot be used as a field type at the class level.

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.

Points to Remember :

1. The variable is a name given to a data value.
2. A variable holds the value of specific data type e.g string, int, float etc.
3. A variable can be declared and initialized later or declared & initialized at the same time.
4. The value of a variable can be changed at any time throughout the program as long as it is accessible.
5. Multiple variables can be defined separated by comma (,) in a single or multiple line till semicolon(;;).
6. A value must be assigned to a variable before using it otherwise it will give compile time error.
7. **var** can only be declared and initialized in a single statement. Following is not valid: `var i; i = 10;`
8. **var** cannot be used as a field type at the class level.
9. **var** cannot be used in an expression like `var i += 10;`
10. Multiple **vars** cannot be declared and initialized in a single statement. For example, `var i=10, j=20;` is invalid.

11. `int i = 100;` // explicitly typed
12. `var i = 100;` // implicitly type

C# Data types:

In the previous section, we have seen that a variable must be declared with the data type because C# is a strongly-typed language. For example,

```
string message = "Hello World!!";
```

string is a data type, message is a variable, and "Hello World!!" is a string value assigned to a variable - message.

The data type tells a C# compiler what kind of value a variable can hold. C# includes many in-built data types for different kinds of data, e.g., String, number, float, decimal, etc.

Each data types includes specific range of values:

Alias	.NET Type	Type	Size (bits)	Range (values)
byte	Byte	Unsigned integer	8	0 to 255
sbyte	SByte	Signed integer	8	-128 to 127
int	Int32	Signed integer	32	-2,147,483,648 to 2,147,483,647
uint	UInt32	Unsigned integer	32	0 to 4294967295
short	Int16	Signed integer	16	-32,768 to 32,767
ushort	UInt16	Unsigned integer	16	0 to 65,535
long	Int64	Signed integer	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	UInt64	Unsigned integer	64	0 to 18,446,744,073,709,551,615
float	Single	Single-precision floating point type	32	-3.402823e38 to 3.402823e38
double	Double	Double-precision floating point type	64	-1.79769313486232e308 to 1.79769313486232e308
char	Char	A single Unicode character	16	Unicode symbols used in text
bool	Boolean	Logical Boolean type	8	True or False
object	Object	Base type of all other types		
string	String	A sequence of characters		
decimal	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	128	(+ or -)1.0 x 10e-28 to 7.9 x 10e28
DateTime	DateTime	Represents date and time		0:00:00am 1/1/01 to 11:59:59pm 12/31/9999

As you can see in the above table that each data types (except string and object) includes value range. Compiler will give an error if value goes out of datatype's permitted range. For example, int data type's range is -2,147,483,648 to 2,147,483,647. So if you assign value which is not in this range then compiler would give error.

Alias vs .Net Type:

In the above table of data types, first column is for data type alias and second column is actual .Net type name. For example, int is an alias (or short name) for Int32. Int32 is a [structure](#) defined in System namespace. The same way, string represent String class.

Alias	Type Name	.Net Type
byte	System.Byte	struct
sbyte	System.SByte	struct
int	System.Int32	struct
uint	System.UInt32	struct
short	System.Int16	struct
ushort	System.UInt16	struct
long	System.Int64	struct
ulong	System.UInt64	struct
float	System.Single	struct
double	System.Double	struct
char	System.Char	struct
bool	System.Boolean	struct
object	System.Object	Class
string	System.String	Class
decimal	System.Decimal	struct
DateTime	System.DateTime	struct

Data types are further classified as *value type* or *reference type*, depending on whether a variable of a particular type stores its own data or a pointer to the data in the memory.

Value Type and Reference Type:

In C#, data types are categorized based on how they store their value in the memory. C# includes following categories of data types:

1. Value type
2. Reference type

Value Type:

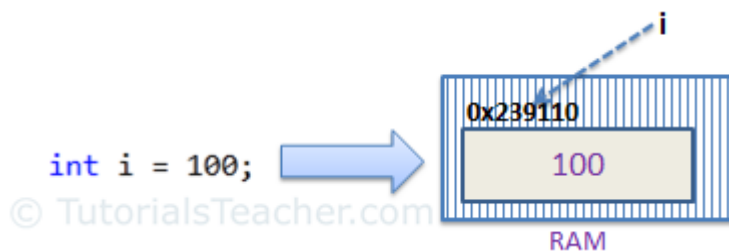
A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values.



All the value types derive from *System.ValueType*, which in-turn, derives from *System.Object*.

For example, consider integer variable `int i = 100;`

The system stores 100 in the memory space allocated for the variable 'i'. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':



The following data types are all of value type:

Bool, byte, char,

decimal, double, enum,

float, int, long, sbyte,

short, struct, uint, ulong,

ushort

Passing by Value:

When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

Example: Value type passes by value

```
static void ChangeValue(int x)
{
    x = 200;

    Console.WriteLine(x);
}

static void Main(string[] args)
{
    int i = 100;

    Console.WriteLine(i);
    //Output: 100

    ChangeValue(i);
    //Output: 200

    Console.WriteLine(i);
    //Output: 100
}
```

In the above example, variable `i` in `Main()` method remains unchanged even after we pass it to the `ChangeValue()` method and change it's value there.

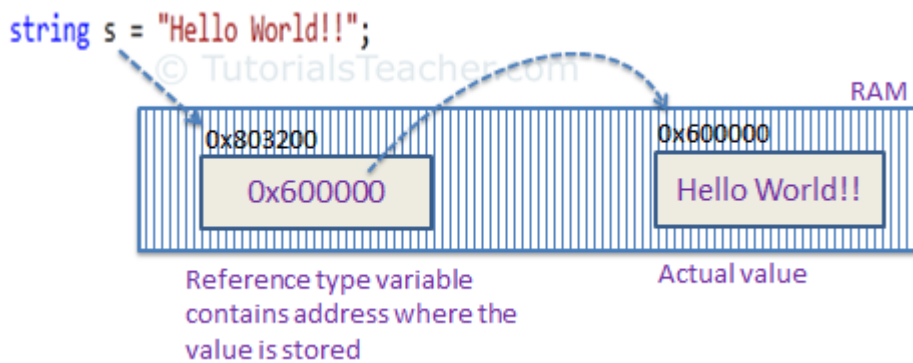
Reference type:

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider following string variable:

```
string s = "Hello World!!";
```

The following image shows how the system allocates the memory for the above string variable.



Memory allocation

for Reference type

As you can see in the above image, the system selects a random location in memory (0x803200) for the variable 's'. The value of a variable s is 0x600000 which is the memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of value itself.

The following data types are of reference type:

- String
- All arrays, even if their elements are value types
- Class
- Delegates

Passing by Reference:

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

Example: Reference type variable passes by reference

```
static void ChangeReferenceType(Student std2)
{
    std2.StudentName = "Steve";
}

static void Main(string[] args)
{
    Student std1 = new Student();
    std1.StudentName = "Bill";

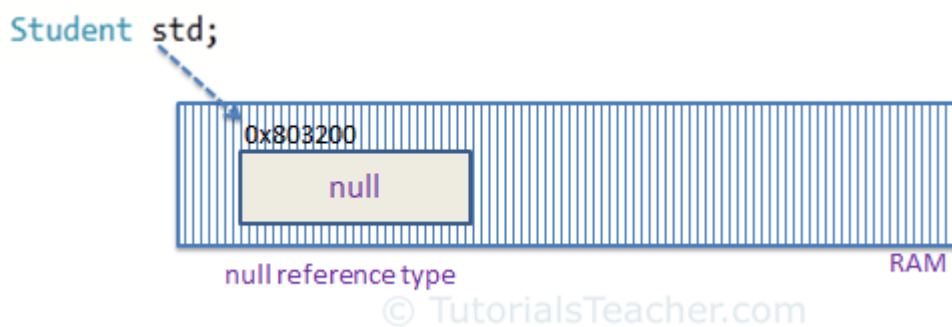
    ChangeReferenceType(std1);

    Console.WriteLine(std1.StudentName);
    //Output: Steve
}
```

In the above example, since Student is an object, when we send the Student object std1 to the ChangeReferenceType() method, what is actually sent is the memory address of std1. Thus, when the ChangeReferenceType() method changes StudentName, it is actually changing StudentName of std1, because std1 and std2 are both pointing to the same address in memory. Therefore, the output is **Steve**.

null value:

Reference types have null value by default, when they are not initialized. For example, a string variable (or any other variable of reference type datatype) without a value assigned to it. In this case, it has a null value, meaning it doesn't point to any other memory location, because it has no value yet.



Null Reference

type

A value type variable cannot be null because it holds a value not a memory address. However, value type variables must be assigned some value before use. The compiler will give an error if you try to use a local value type variable without assigning a value to it.

Example: Compile time error

```
void someFunction()
{
    int i;
    Console.WriteLine(i);
}
```



C# 2.0 introduced nullable types for value types so that you can assign null to a value type variable or declare a value type variable without assigning a value to it.

However, value type field in a class can be declared without initialization (field not a local variable in the function) . It will have a default value if not assigned any value, e.g., int will have 0, boolean will have false and so on.

Example: Value type field

```
class myClass
{
    public int i;
}

myClass mcls = new myClass();

Console.WriteLine(mcls.i);
//Output: 0
```

Visit MSDN to read about [default values of value types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table).

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table>

Points to Remember :

1. Value type stores the value in its memory space, whereas reference type stores the address of the value where it is stored.
2. Primitive data types and struct are of the 'Value' type. Class objects, string, array, delegates are reference types.
3. Value type passes byval by default. Reference type passes byref by default.
4. **Value types** and **reference types** stored in **Stack** and **Heap** in the memory depends on the scope of the variable.

C# Keywords:

C# contains reserved words, that have special meaning for the compiler. These reserved words are called "keywords". Keywords cannot be used as a name (identifier) of a variable, class, interface, etc.

Keywords in C# are distributed under the following categories:

Modifier keywords

Modifier keywords are certain keywords that indicate who can modify types and type members. Modifiers allow or prevent certain parts of programs from being modified by other parts.

Modifier keywords	
abstract	
async	
const	
event	
extern	
new	
override	
partial	
readonly	
sealed	
static	
unsafe	
virtual	
volatile	

Access Modifier Keywords:

Access modifiers are applied on the declaration of the class, method, properties, fields and other members. They define the accessibility of the class and its members. The default access modifier **for class and struct** is **private** and **for enum and interface** is **public**.

Access Modifiers	Usage
public	The Public modifier allows any part of the program in the same assembly or another assembly to access the type and its members.
private	The Private modifier restricts other parts of the program from accessing the type and its members. Only code in the same class or struct can access it.
internal	The Internal modifier allows other program code in the same assembly to access the type or its members.
protected	The Protected modifier allows codes in the same class or a class that derives from that class to access the type or its members.

Statement Keywords:

Statement keywords are related to program flow.

Statement Keywords	
if	
else	
switch	
case	
do	
for	
foreach	
in	
while	
break	
continue	
default	
goto	
return	
yield	
throw	
try	
catch	
finally	
checked	
unchecked	
fixed	
lock	

Method parameter keywords:

These keywords are applied on the parameters of a method.

Method Parameter Keywords	
params	
ref	
out	

Namespace keywords:

These keywords are applied with namespace and related operators.

Namespace Keywords	
using	
. operator	
:: operator	
extern alias	

Operator Keywords:

Operator keywords perform miscellaneous actions.

Operator Keywords	
as	
await	
is	
new	
sizeof	
typeof	
stackalloc	
checked	
unchecked	

Access keywords:

Access keywords are used to access the containing class or the base class of an object or class.

Access keywords	
base	
this	

Literal keywords:

Literal keywords apply to the current instance or value of an object.

Literal Keywords	
null	
FALSE	
TRUE	
value	
void	

Type keywords:

Type keywords are used for data types.

Type keywords	
bool	
byte	
char	
class	
decimal	
double	
enum	
float	
int	
long	
sbyte	
short	
string	
struct	
uint	
ulong	
ushort	

Contextual Keywords:

Contextual keywords are considered as keywords, only if used in certain contexts. They are not reserved and so can be used as names or identifiers.

Contextual Keywords	
add	
var	
dynamic	
global	
set	
value	

Contextual keywords are not converted into blue color (default color for keywords in visual studio) when used as an identifier in Visual Studio. For

example, var in the below figure is not in blue color whereas color of this is blue color. So var is a contextual keyword.

```
public class @class
{
    public static int var { get; set; }
    public static int this { get; set; }
}
```

Reserved keyword used with @ prefix

Contextual keyword

Reserved keyword

© TutorialsTeacher.com

Query keywords:

Query keywords are contextual keywords used in LINQ queries.

Query Keywords	
from	
where	
select	
group	
into	
orderby	
join	
let	
in	
on	
equals	
by	
ascending	
descending	

As mentioned above, **keyword cannot be used as an identifier** (name of variable, class, interface etc). However, they can be used with the prefix '@'. For example, class is a reserved keyword so it cannot be used as an identifier, but @class can be used as shown below.

Example: Keyword as identifier

```
public class @class
{
    public static int MyProperty { get; set; }
}

@class.MyProperty = 100;
```

Points to Remember :

1. Keywords are reserved words that cannot be used as name or identifier.
2. Prefix '@' with keywords if you want to use it as identifier.
3. C# includes various categories of keywords e.g. modifier keywords, access modifiers keywords, statement keywords, method param keywords etc.
4. Contextual keywords can be used as identifier.

Inheritance & Polymorphism

Inheritance

Inheritance allows us to define a class based on another class. This makes creating and maintaining an application easy.

The class whose properties are inherited by another class is called **the Base class**. The class which inherits the properties is called **the Derived class**.

For example, base class **Animal** can be used to derive **Cat** and **Dog** classes. The derived class inherits all the features from the base class, and can have its own additional features.

```
class Animal {
    public int Legs {get; set;}
    public int Age {get; set;}
}
class Dog : Animal {
    public Dog() {
        Legs = 4;
    }
    public void Bark() {
        Console.WriteLine("Woof");
    }
}
static void Main(string[] args)
{
    Dog d = new Dog();
    Console.WriteLine(d.Legs);

    d.Bark();
}
```

Note the syntax for a derived class. A **colon** and the name of the **base** class follow the name of the derived class.

All **public** members of **Animal** become **public** members of **Dog**. That is why we can access the **Legs** member in the **Dog constructor**.

Now we can instantiate an object of type **Dog** and access the inherited members as well as call its own **Bark method**.

A base class can have multiple derived classes. For example, a **Cat** class can inherit from **Animal**.

Inheritance allows the derived class to reuse the code in the base class without having to rewrite it. And the derived class can be customized by

adding more members. In this manner, the derived class extends the functionality of the base class.

A derived class inherits all the members of the base class, including its methods.

For example:

```
class Program
{
    class Person {
        public void Speak() {
            Console.WriteLine("Hi there");
        }
    }
    class Student : Person {
        int number;
    }

    static void Main(string[] args)
    {
        Student s = new Student();
        s.Speak();
    }
}
```

We created a Student object and called the Speak method, which was declared in the base class Person.

C# does not support multiple inheritance, so you cannot inherit from multiple classes. However, you can use interfaces to implement multiple inheritance. You will learn more about interfaces in the coming lessons.

Protected Members

Protected

Up to this point, we have worked exclusively with **public** and **private** access modifiers.

Public members may be accessed from anywhere outside of the class, while access to **private** members is limited to their class.

The **protected** access modifier is very similar to **private** with one difference; it can be accessed in the derived classes. So, a **protected** member is accessible only from derived classes.

For example:

```
class Program
{
    class Person {
        protected int Age {get; set;}
        protected string Name {get; set;}
    }
    class Student : Person {
        public Student(string nm) {
            Name = nm;
        }
        public void Speak() {
            Console.WriteLine("Name: "+Name);
        }
    }
    static void Main(string[] args)
    {
        Student s = new Student("David");
        s.Speak();
        //Output: "Name: David"
    }
}
```

As you can see, we can access and modify the **Name** property of the base class from the derived class.

But, if we try to access it from outside code, we will get an error:

```
static void Main(string[] args)
{
    Student s = new Student("David");
    s.Name = "Bob"; //Error
}
```

Sealed

A class can prevent other classes from inheriting it, or any of its members, by using the **sealed** modifier.

For example:

```
sealed class Animal {  
    //some code  
}  
class Dog : Animal { } //Error
```

In this case, we cannot derive the Dog class from the Animal class because Animal is **sealed**.

The **sealed** keyword provides a level of protection to your class so that other classes cannot inherit from it.

Derived Class Constructor & Destructor

Constructors are called when objects of a class are created. With [inheritance](#), the base class [constructor](#) and [destructor](#) are not inherited, so you should define constructors for the derived classes.

However, the base class [constructor](#) and [destructor](#) are being invoked automatically when an object of the derived class is created or deleted. Consider the following example:

```
class Program
{
    class Animal {
        public Animal() {
            Console.WriteLine("Animal created");
        }
        ~Animal() {
            Console.WriteLine("Animal deleted");
        }
    }
    class Dog: Animal {
        public Dog() {
            Console.WriteLine("Dog created");
        }
        ~Dog() {
            Console.WriteLine("Dog deleted");
        }
    }
    static void Main(string[] args)
    {
        Dog d = new Dog();
    }
}
```

We have defined the Animal class with a [constructor](#) and [destructor](#) and a derived Dog class with its own [constructor](#) and [destructor](#).

So what will happen when we create an object of the derived class?

Outputs:

```
Animal created
Dog created
Dog deleted
Animal deleted
```

Note that the base class constructor is called first and the derived class constructor is called next.

When the object is destroyed, the derived class destructor is invoked and then the base class destructor is invoked.

You can think of it as the following: The derived class needs its base class in order to work, which is why the base class constructor is called first.

Polymorphism

The word **polymorphism** means "having many forms".

Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related through **inheritance** from a common base class.

Polymorphism means that a call to a member **method** will cause a different implementation to be executed depending on the **type** of object that invokes the **method**.

Simply, **polymorphism** means that a single **method** can have a number of different implementations.

Consider having a program that allows users to draw different shapes. Each shape is drawn differently, and you do not know which shape the user will choose.

Here, polymorphism can be leveraged to invoke the appropriate Draw method of any derived class by overriding the same method in the base class. Such methods must be declared using the virtual keyword in the base class.

For example:

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Base Draw");
    }
}
```

The virtual keyword allows methods to be overridden in derived classes.

Virtual methods enable you to work with groups of related objects in a uniform way.

Now, we can derive different shape classes that define their own Draw methods using the **override** keyword:

```

class Circle : Shape
{
    public override void Draw()
    {
        // draw a circle...
        Console.WriteLine("Circle Draw");
    }
}

class Rectangle : Shape
{
    public override void Draw()
    {
        // draw a rectangle...
        Console.WriteLine("Rect Draw");
    }
}

```

The virtual Draw method in the Shape base class can be overridden in the derived classes. In this case, Circle and Rectangle have their own Draw methods.

Now, we can create separate Shape objects for each derived type and then call their Draw methods:

```

static void Main(string[] args)
{
    Shape c = new Circle();
    c.Draw();
    //Outputs "Circle Draw"

    Shape r = new Rectangle();
    r.Draw();
    //Outputs "Rect Draw"
}

```

As you can see, each object invoked its own Draw method, thanks to polymorphism.

To summarize, **polymorphism** is a way to call the same **method** for different objects and generate different results based on the object type. This behavior is achieved through virtual methods in the base class.

To implement this, we create objects of the base type, but instantiate them as the derived type:

```
Shape c = new Circle();
```

Shape is the **base class**. **Circle** is the **derived class**.

So why use **polymorphism**? We could just instantiate each object of its type and call its **method**, as in:

```
Circle c = new Circle();  
c.Draw();
```

The polymorphic approach allows us to treat each object the same way. As all objects are of type Shape, it is easier to maintain and work with them. You could, for example, have a list (or **array**) of objects of that type and work with them dynamically, without knowing the actual derived type of each object.

Polymorphism can be useful in many cases. For example, we could create a game where we would have different Player types with each Player having a separate behavior for the Attack method.

In this case, Attack would be a virtual method of the base class Player and each derived class would override it.

