Anonymous Types

So far, we have learned that objects come from a class. A class is declared with a number of fields, properties and/or methods, and then you can create an instance of this class as an object. However, with the introduction of anonymous types, you no longer have to declare a class before creating an object. Don't worry, classes are not dead at all, because anonymous types (or objects) comes with several limitations, but for some situations, they're really great!

An anonymous type is initialized using the *new* operator, in combination with an object initializer - in that regard, it's very much like instantiating a class, only you leave out the class name. Also, since there's no class behind the object, you must use the *var* keyword when retrieving the reference to your object. This might sound complicated, but the following example should demonstrate to you that it's not complicated at all:

var user = new      
{      
 Name = "John Doe",      
 Age = 42     
};  
Console.WriteLine(user.Name + " - " + user.Age + " years old");

That's it - we now have an object with information (name and age) about a user. Anonymous types are great for a lot of situations, especially when you just need to return something quickly, with more complexity than just a string or a number. Anonymous types allow you to make up properties on the fly, without worrying about declaring a class first and then alter this class when your need changes. But as mentioned, there are several limitations that you need to be aware of when you consider using an anonymous type over defining a class:

* Unlike a real class, an anonymous type can't have fields or methods - only properties
* Once the object has been initialized, you can't add new properties to it
* Properties are readonly - as soon as the object has been initialized, you can't change their values

But with that said, anonymous types are still extremely practical for a lot of tasks. A common usage scenario is when you have a complex object (from a defined class) and you need to simplify it, e.g. because you have to keep the object as small as possible to send it to a browser or perhaps because the full object has sensitive information that you don't want to expose to the consumer. Anonymous types are great for this, as illustrated in this next example:

sing System;  
using System.IO;  
  
namespace AnonymousTypes  
{  
    class Program  
    {  
 static void Main(string[] args)  
 {  
     string pathOfExe = System.Reflection.Assembly.GetEntryAssembly().Location;  
     FileInfo fileInfo = new FileInfo(pathOfExe);  
     var simpleFileInfo = new  
     {  
 Filename = fileInfo.Name,  
 FileSize = fileInfo.Length  
     };  
     Console.WriteLine("File name: " + simpleFileInfo.Filename + ". Size: " + simpleFileInfo.FileSize + " bytes");  
 }  
    }  
}

static void Main(string[] args)

{

var myAnonymousType = new { firstProperty = "First",

secondProperty = 2,

thirdProperty = true

};

Console.WriteLine(myAnonymousType.GetType().ToString());

}

## Nested Anonymous Type

An anonymous type can have another anonymous type as a property.

Example: Nested Anonymous Type

var myAnonymousType = new

{

firstProperty = "First",

secondProperty = 2,

thirdProperty = true,

anotherAnonymousType = new { nestedProperty = "Nested"}

};

## Scope of Anonymous Type

An anonymous type will always be local to the method where it is defined. Usually, you cannot pass an anonymus type to another method; however, you can pass it to a method that accepts a parameter of [dynamic type](https://www.tutorialsteacher.com/csharp/csharp-dynamic-type). Please note that Passing anonymous types using dynamic is not recommended.

Example: Passing Anonymous Type

static void Main(string[] args)

{

var myAnonymousType = new

{

firstProperty = "First Property",

secondProperty = 2,

thirdProperty = true

};

DoSomethig(myAnonymousType);

}

static void DoSomethig(dynamic param)

{

Console.WriteLine(param.firstProperty);

}

Output:

First Property

1. Anonymous type can be defined using the new keyword and object initializer syntax.
2. The implicitly typed variable- **var**, is used to hold an anonymous type.
3. Anonymous type is a **reference type** and all the properties are read-only.
4. The scope of an anonymous type is local to the method where it is defined.

C# - Dynamic Type

C# 4.0 (.NET 4.5) introduced a new type that avoids compile time type checking. You have learned about the implicitly typed variable- [var](https://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable) in the previous section where the compiler assigns a specific type based on the value of the expression. A dynamic type escapes type checking at compile time; instead, it resolves type at run time.

A dynamic type can be defined using the dynamic keyword.

1. The dynamic types are resolved at runtime instead of compile time.
2. The compiler skips the type checking for dynamic type. So it doesn't give any error about dynamic types at compile time.
3. The dynamic types do not have intellisense support in visual studio.
4. A method can have parameters of the dynamic type.
5. An exception is thrown at runtime if a method or property is not compatible.

Example: dynamic type variable

dynamic dynamicVariable = 1;

The compiler compiles dynamic types into object types in most cases. The above statement would be compiled as:

dynamic type at compile time:

object dynamicVariable = 1;

The actual type of dynamic would resolve at runtime. You can check the type of the dynamic variable, as below:

Example: Get the actual type of dynamic type at runtime

static void Main(string[] args)

{

dynamic dynamicVariable = 1;

Console.WriteLine(dynamicVariable.GetType().ToString());

}

Output:

System

Static void Main(string[] args)

{

dynamic dynamicVariable = 100;

Console.WriteLine("Dynamic variable value: {0}, Type: {1}",dynamicVariable, dynamicVariable.GetType().ToString());

dynamicVariable = "Hello World!!";

Console.WriteLine("Dynamic variable value: {0}, Type: {1}", dynamicVariable, dynamicVariable.GetType().ToString());

dynamicVariable = true;

Console.WriteLine("Dynamic variable value: {0}, Type: {1}", dynamicVariable, dynamicVariable.GetType().ToString());

dynamicVariable = DateTime.Now;

Console.WriteLine("Dynamic variable value: {0}, Type: {1}", dynamicVariable, dynamicVariable.GetType().ToString());

}

.Int32

## Methods and Properties of Dynamic Type

If you assign class object to the dynamic type then the compiler would not check for correct methods and properties name of a dynamic type that holds the custom class object. Consider the following example.

Example: dynamic

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

public int StandardID { get; set; }

public void DisplayStudentDetail()

{

Console.WriteLine("Name: {0}", this.StudentName);

Console.WriteLine("Age: {0}", this.Age);

Console.WriteLine("Standard: {0}", this.StandardID);

}

}

class Program

{

static void Main(string[] args)

{

dynamic dynamicStudent = new Student();

dynamicStudent.FakeMethod();

}

}

In the above example, we have assigned Student object to a dynamic variable. In the second statement in Main() method, we call FakeMethod() method, which is not exists in the Student class. However, the compiler will not give any error for FakeMethod() because it skips type checking for dynamic type, instead you will get a runtime exception

## Dynamic Type as a Method Parameter

A method can have dynamic type parameters so that it can accept any type of parameter at run time.

Example: dynamic as a Parameter

class Program

{

static void PrintValue(dynamic val)

{

Console.WriteLine(val);

}

static void Main(string[] args)

{

PrintValue("Hello World!!");

PrintValue(100);

PrintValue(100.50);

PrintValue(true);

PrintValue(DateTime.Now);

}

}