**From tutorialspoint very Basic you know that shit.**

**C# Destructors**

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope. A **destructor** has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.

Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

Following example explains the concept of destructor –

**class Program**

**{**

**private double length;**

**public Program() // constructor**

**{**

**Console.WriteLine("Object is being created");**

**}**

**static void Main(string[] args)**

**{**

**Program program = new Program();**

**program.setLength(6.0);**

**Console.WriteLine("Length of line : {0}", program.getLength());**

**}**

**~Program() //destructor**

**{**

**Console.WriteLine("Object is being deleted");**

**}**

**public void setLength(double len)**

**{**

**length = len;**

**}**

**public double getLength()**

**{**

**return length;**

**}**

**}**

**}**

Object is being created

Length of line : 6

Object is being deleted

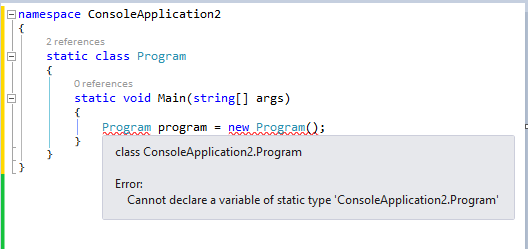
## Static Members of a C# Class

The keyword **static** implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.

The following example demonstrates the use of **static variables** –

# **Classes (C# Programming Guide)**

 If the class is not declared as static, client code can create instances of it. These instances are objects which are assigned to a variable. The instance of a class remains in memory until all references to it go out of scope. At that time, the CLR marks it as eligible for garbage collection. If the class is declared as [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static), you cannot create instances, and client code can only access it through the class itself. For more information, see [Static Classes and Static Class Members](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members).



**Reference types**

A type that is defined as a [class](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class) is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value [null](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/null) until you explicitly create an instance of the class by using the [new](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/new) operator, or assign it an object that has been created elsewhere, as shown in the following example

MyClass mc = new MyClass();

MyClass mc2 = mc;

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as garbage collection. However, garbage collection is also highly optimized, and in most scenarios, it does not create a performance issue. For more information about garbage collection, see [Automatic memory management and garbage collection](https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/gc).

## Class Inheritance

When you create a class, you can inherit from any other interface or class that is not defined as [sealed](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed), and other classes can inherit from your class and override class virtual methods.

Inheritance is accomplished by using a derivation, which means a class is declared by using a base class from which it inherits data and behavior. A base class is specified by appending a colon and the name of the base class following the derived class name, like this:

public class Manager : Employee

{

// Employee fields, properties, methods and events are inherited

// New Manager fields, properties, methods and events go here...

}

When a class declares a base class, it inherits all the members of the base class except the constructors. For more information, see [Inheritance](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance).

Unlike C++, a class in C# can only directly inherit from one base class. However, because a base class may itself inherit from another class, a class may indirectly inherit multiple base classes. Furthermore, a class can directly implement more than one interface. For more information, see [Interfaces](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/index).

A class can be declared [abstract](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract). An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods. By contrast, a [sealed](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed) class does not allow other classes to derive from it. For more information, see [Abstract and Sealed Classes and Class Members](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members).

**public class Person**

**{**

**// Constructor that takes no arguments:**

**public Person()**

**{**

**Name = "unknown";**

**}**

**// Constructor that takes one argument:**

**public Person(string name)**

**{**

**Name = name;**

**}**

**// Auto-implemented readonly property:**

**public string Name { get; }**

**// Method that overrides the base class (System.Object) implementation.**

**public override string ToString()**

**{**

**return Name.ToString();**

**}**

**}**

**class TestPerson**

**{**

**static void Main()**

**{**

**// Call the constructor that has no parameters.**

**var person1 = new Person();**

**Console.WriteLine(person1.Name);**

**// Call the constructor that has one parameter.**

**var person2 = new Person("Sarah Jones");**

**Console.WriteLine(person2.Name);**

**// OR use ToString method to get string representation of the person2 instance:**

**Console.WriteLine(person2.ToString());**

**Console.WriteLine("Press any key to exit.");**

**Console.ReadKey();**

**}**

**}**

**// Output:**

**// unknown**

**// Sarah Jones**

**// Sarah Jones**