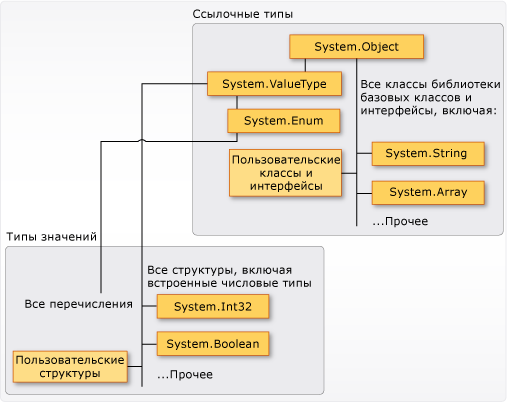
Наследование

* Напоминание! Все объекты наследуются от Object



[Ссылка с которой начать](https://msdn.microsoft.com/ru-ru/library/ms173149.aspx)

1. Базовый класс (private, protected, public):
2. [public](https://msdn.microsoft.com/ru-ru/library/yzh058ae.aspx)  
   Доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него.
3. [private](https://msdn.microsoft.com/ru-ru/library/st6sy9xe.aspx)  
   Доступ к типу или члену можно получить только из кода в том же классе или структуре.
4. [protected](https://msdn.microsoft.com/ru-ru/library/bcd5672a.aspx)  
   Доступ к типу или элементу можно получить только из кода в том же классе или структуре, либо в производном классе.
5. [internal](https://msdn.microsoft.com/ru-ru/library/7c5ka91b.aspx)  
   Доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.

// WorkItem implicitly inherits from the Object class.

public class WorkItem

{

// Static field currentID stores the job ID of the last WorkItem that

// has been created.

private static int currentID;

//Properties.

protected int ID { get; set; }

protected string Title { get; set; }

protected string Description { get; set; }

protected TimeSpan jobLength { get; set; }

// Default constructor. If a derived class does not invoke a base-

// class constructor explicitly, the default constructor is called

// implicitly.

public WorkItem()

{

ID = 0;

Title = "Default title";

Description = "Default description.";

jobLength = new TimeSpan();

}

// Instance constructor that has three parameters.

public WorkItem(string title, string desc, TimeSpan joblen)

{

this.ID = GetNextID();

this.Title = title;

this.Description = desc;

this.jobLength = joblen;

}

// Static constructor to initialize the static member, currentID. This

// constructor is called one time, automatically, before any instance

// of WorkItem or ChangeRequest is created, or currentID is referenced.

static WorkItem()

{

currentID = 0;

}

protected int GetNextID()

{

// currentID is a static field. It is incremented each time a new

// instance of WorkItem is created.

return ++currentID;

}

// Method Update enables you to update the title and job length of an

// existing WorkItem object.

public void Update(string title, TimeSpan joblen)

{

this.Title = title;

this.jobLength = joblen;

}

// Virtual method override of the ToString method that is inherited

// from System.Object.

public override string ToString()

{

return String.Format("{0} - {1}", this.ID, this.Title);

}

}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)

// and two constructors.

public class ChangeRequest : WorkItem

{

protected int originalItemID { get; set; }

// Constructors. Because neither constructor calls a base-class

// constructor explicitly, the default constructor in the base class

// is called implicitly. The base class must contain a default

// constructor.

// Default constructor for the derived class.

public ChangeRequest() { }

// Instance constructor that has four parameters.

public ChangeRequest(string title, string desc, TimeSpan jobLen,

int originalID)

{

// The following properties and the GetNexID method are inherited

// from WorkItem.

this.ID = GetNextID();

this.Title = title;

this.Description = desc;

this.jobLength = jobLen;

// Property originalItemId is a member of ChangeRequest, but not

// of WorkItem.

this.originalItemID = originalID;

}

}

class Program

{

static void Main()

{

// Create an instance of WorkItem by using the constructor in the

// base class that takes three arguments.

WorkItem item = new WorkItem("Fix Bugs",

"Fix all bugs in my code branch",

new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in

// the derived class that takes four arguments.

ChangeRequest change = new ChangeRequest("Change Base Class Design",

"Add members to the class",

new TimeSpan(4, 0, 0),

1);

// Use the ToString method defined in WorkItem.

Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the

// ChangeRequest object.

change.Update("Change the Design of the Base Class",

new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.

Console.WriteLine(change.ToString());

// Keep the console open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

1 - Fix Bugs

2 - Change the Design of the Base Class

\*/

* Нет множественного наследования!
* Возможен только 1 базовый класс

1. Абстрактный класс:

public abstract class A

{

public abstract void DoWork(int i);

}

* Создать такой класс нельзя
* Необходимо имплементировать все абстрактные методы

1. Запечатанный класс и метод:

public sealed class D

{

// Class members here.

}

* Наследоваться от такого класса нельзя

public class D : C

{

public sealed override void DoWork() { }

}

* Переопределить такой метод нельзя

1. Интерфейс:

interface ISampleInterface

{

void SampleMethod();

}

class ImplementationClass : ISampleInterface

{

// Explicit interface member implementation:

void ISampleInterface.SampleMethod()

{

// Method implementation.

}

static void Main()

{

// Declare an interface instance.

ISampleInterface obj = new ImplementationClass();

// Call the member.

obj.SampleMethod();

}

}

1. Универсальные классы и интерфейсы

// Declare the generic class.

public class GenericList<T>

{

void Add(T input) { }

}

class TestGenericList

{

private class ExampleClass { }

static void Main()

{

// Declare a list of type int.

GenericList<int> list1 = new GenericList<int>();

// Declare a list of type string.

GenericList<string> list2 = new GenericList<string>();

// Declare a list of type ExampleClass.

GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();

}

}

public interface ISessionChannel<TSession>

{

TSession Session { get; }

}

* Универсальные классы и универсальные интерфейсы != шаблоны

И вот почему:

|  |  |
| --- | --- |
| Ограничение | Описание |
| where T: struct | Аргумент типа должен иметь тип значения. Допускается указание любого типа значения, кроме Nullable. Дополнительные сведения см. в разделе Использование допускающих значение NULL типов. |
| where T : class | Аргумент типа должен иметь ссылочный тип; это также распространяется на тип любого класса, интерфейса, делегата или массива. |
| where T : new() | Аргумент типа должен иметь открытый конструктор без параметров. При использовании с другими ограничениями ограничение new() должно устанавливаться последним. |
| where T : <base class name> | Аргумент типа должен являться или быть производным от указанного базового класса. |
| where T : <interface name> | Аргумент типа должен являться или реализовывать указанный интерфейс. Можно установить несколько ограничений интерфейса. Ограничивающий интерфейс также может быть универсальным. |
| where T : U | Аргумент типа, предоставляемый в качестве T, должен совпадать с аргументом, предоставляемым в качестве U, или быть производным от него. |

В итоге:

* Универсальные шаблоны языка C# не обеспечивают такую же гибкость, как шаблоны языка C++. Например, невозможно вызвать арифметические операторы в универсальном классе языка C#, хотя можно вызвать операторы, определенные пользователем.
* Язык C# не позволяет использовать не являющиеся типами параметры шаблона, например template C<int i> {}.
* Язык C# не поддерживает явную специализацию, т.е. индивидуальную реализацию шаблона для конкретного типа.
* Язык C# не поддерживает частичную специализацию: индивидуальную реализацию для подмножества аргументов типа.
* Язык C# не позволяет использовать параметр типа в качестве базового класса для универсального типа.
* Язык C# не позволяет параметрам типов иметь типы по умолчанию.
* В языке C# параметр универсального типа не может сам быть универсального типа, хотя сконструированные типы могут использоваться как универсальные. Язык C++ не допускает использование параметров шаблона.
* Язык C++ позволяет использование кода, который может быть допустимым не для всех параметров типа в шаблоне и который затем проверяется для конкретного типа, используемого в качестве параметра типа. В языке C# код класса должен быть написан таким образом, чтобы он работал с любым типом, удовлетворяющим требования существующих ограничений. Например, в языке C++ можно написать функцию, которая использует арифметические операторы + и - применительно к объектам параметра типа, что приведет к ошибке во время создания экземпляра шаблона с типом, который не поддерживает эти операторы. Язык C# не позволяет это делать. Допускаются только те языковые конструкции, которые могут быть выведены из ограничений.

Пример:

public class Employee

{

private string name;

private int id;

public Employee(string s, int i)

{

name = s;

id = i;

}

public string Name

{

get { return name; }

set { name = value; }

}

public int ID

{

get { return id; }

set { id = value; }

}

}

public class GenericList<T> where T : Employee

{

private class Node

{

private Node next;

private T data;

public Node(T t)

{

next = null;

data = t;

}

public Node Next

{

get { return next; }

set { next = value; }

}

public T Data

{

get { return data; }

set { data = value; }

}

}

private Node head;

public GenericList() //constructor

{

head = null;

}

public void AddHead(T t)

{

Node n = new Node(t);

n.Next = head;

head = n;

}

public IEnumerator<T> GetEnumerator()

{

Node current = head;

while (current != null)

{

yield return current.Data;

current = current.Next;

}

}

public T FindFirstOccurrence(string s)

{

Node current = head;

T t = null;

while (current != null)

{

//The constraint enables access to the Name property.

if (current.Data.Name == s)

{

t = current.Data;

break;

}

else

{

current = current.Next;

}

}

return t;

}

}

Аналогично для интерфейсов:

class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()

{

// ...

}

Для нескольких параметров:

class Base { }

class Test<T, U>

where U : struct

where T : Base, new() { }

1. virtual new override

class Program

{

static void Main(string[] args)

{

// Declare objects of the derived classes and test which version

// of ShowDetails is run, base or derived.

TestCars1();

// Declare objects of the base class, instantiated with the

// derived classes, and repeat the tests.

TestCars2();

// Declare objects of the derived classes and call ShowDetails

// directly.

TestCars3();

// Declare objects of the base class, instantiated with the

// derived classes, and repeat the tests.

TestCars4();

}

public static void TestCars1()

{

System.Console.WriteLine("\nTestCars1");

System.Console.WriteLine("----------");

Car car1 = new Car();

car1.DescribeCar();

System.Console.WriteLine("----------");

// Notice the output from this test case. The new modifier is

// used in the definition of ShowDetails in the ConvertibleCar

// class.

ConvertibleCar car2 = new ConvertibleCar();

car2.DescribeCar();

System.Console.WriteLine("----------");

Minivan car3 = new Minivan();

car3.DescribeCar();

System.Console.WriteLine("----------");

}

// Output:

// TestCars1

// ----------

// Four wheels and an engine.

// Standard transportation.

// ----------

// Four wheels and an engine.

// Standard transportation.

// ----------

// Four wheels and an engine.

// Carries seven people.

// ----------

public static void TestCars2()

{

System.Console.WriteLine("\nTestCars2");

System.Console.WriteLine("----------");

var cars = new List<Car> { new Car(), new ConvertibleCar(),

new Minivan() };

foreach (var car in cars)

{

car.DescribeCar();

System.Console.WriteLine("----------");

}

}

// Output:

// TestCars2

// ----------

// Four wheels and an engine.

// Standard transportation.

// ----------

// Four wheels and an engine.

// Standard transportation.

// ----------

// Four wheels and an engine.

// Carries seven people.

// ----------

public static void TestCars3()

{

System.Console.WriteLine("\nTestCars3");

System.Console.WriteLine("----------");

ConvertibleCar car2 = new ConvertibleCar();

Minivan car3 = new Minivan();

car2.ShowDetails();

car3.ShowDetails();

}

// Output:

// TestCars3

// ----------

// A roof that opens up.

// Carries seven people.

public static void TestCars4()

{

System.Console.WriteLine("\nTestCars4");

System.Console.WriteLine("----------");

Car car2 = new ConvertibleCar();

Car car3 = new Minivan();

car2.ShowDetails();

car3.ShowDetails();

}

// Output:

// TestCars4

// ----------

// Standard transportation.

// Carries seven people.

}

// Define the base class, Car. The class defines two virtual methods,

// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived

// class also defines a ShowDetails method. The example tests which version of

// ShowDetails is used, the base class method or the derived class method.

class Car

{

public virtual void DescribeCar()

{

System.Console.WriteLine("Four wheels and an engine.");

ShowDetails();

}

public virtual void ShowDetails()

{

System.Console.WriteLine("Standard transportation.");

}

}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails

// hides the base class method.

class ConvertibleCar : Car

{

public new void ShowDetails()

{

System.Console.WriteLine("A roof that opens up.");

}

}

// Class Minivan uses the override modifier to specify that ShowDetails

// extends the base class method.

class Minivan : Car

{

public override void ShowDetails()

{

System.Console.WriteLine("Carries seven people.");

}

}

* По умолчанию new (warning при компиляции)
* Обычно не стоит использовать New
* Базовую реализация можно вызвать, используя base.

Реализация 2-х интерфейсов с одним именем:

class Test

{

static void Main()

{

SampleClass sc = new SampleClass();

IControl ctrl = (IControl)sc;

ISurface srfc = (ISurface)sc;

// The following lines all call the same method.

sc.Paint();

ctrl.Paint();

srfc.Paint();

}

}

interface IControl

{

void Paint();

}

interface ISurface

{

void Paint();

}

class SampleClass : IControl, ISurface

{

// Both ISurface.Paint and IControl.Paint call this method.

public void Paint()

{

Console.WriteLine("Paint method in SampleClass");

}

}

// Output:

// Paint method in SampleClass

// Paint method in SampleClass

// Paint method in SampleClass

Различная реализация:

public class SampleClass : IControl, ISurface

{

void IControl.Paint()

{

System.Console.WriteLine("IControl.Paint");

}

void ISurface.Paint()

{

System.Console.WriteLine("ISurface.Paint");

}

}

SampleClass sc = new SampleClass();

Sc.Paint() //<- error

Свойства в интерфейсе:

interface IEmployee

{

string Name

{

get;

set;

}

int Counter

{

get;

}

}

public class Employee : IEmployee

{

public static int numberOfEmployees;

private string name;

public string Name // read-write instance property

{

get

{

return name;

}

set

{

name = value;

}

}

private int counter;

public int Counter // read-only instance property

{

get

{

return counter;

}

}

public Employee() // constructor

{

counter = ++counter + numberOfEmployees;

}

}

class TestEmployee

{

static void Main()

{

System.Console.Write("Enter number of employees: ");

Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();

System.Console.Write("Enter the name of the new employee: ");

e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");

System.Console.WriteLine("Employee number: {0}", e1.Counter);

System.Console.WriteLine("Employee name: {0}", e1.Name);

}

}

//Enter number of employees: 210

//Enter the name of the new employee: Hazem Abolrous

//The employee information:

//Employee number: 211

//Employee name: Hazem Abolrous

1. Делегаты

Предназначение аналогично указателю на метод в с++

Объявление:

public delegate void Del(string message);

Реализация:

// Create a method for a delegate.

public static void DelegateMethod(string message)

{

System.Console.WriteLine(message);

}

Использование:

// Instantiate the delegate.

Del handler = DelegateMethod;

// Call the delegate.

handler("Hello World");

Делегат – объект, наследуется от Delegate, является запечатанными.

Соответственно:

* типо-безопасность
* передача в методы
* может являться свойством

Могут быть шаблонными:

delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>

{

public class StackEventArgs : System.EventArgs { }

public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

protected virtual void OnStackChanged(StackEventArgs a)

{

stackEvent(this, a);

}

}

class SampleClass

{

public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }

}

public static void Test()

{

Stack<double> s = new Stack<double>();

SampleClass o = new SampleClass();

s.stackEvent += o.HandleStackChange;

}