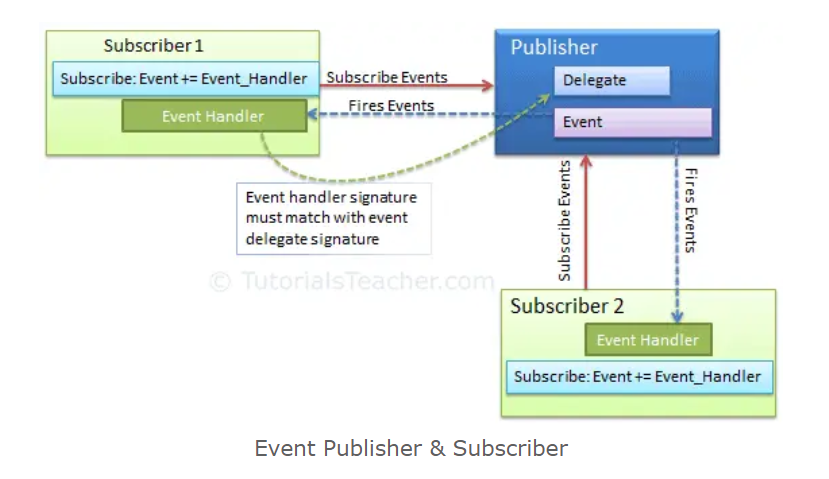
Event:

An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the [observer design pattern](https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern" \t "https://www.tutorialsteacher.com/csharp/_blank).

The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.

In C#, an event is an encapsulated [delegate](https://www.tutorialsteacher.com/csharp/csharp-delegates). It is dependent on the delegate. The [delegate](https://www.tutorialsteacher.com/csharp/csharp-delegates) defines the signature for the event handler method of the subscriber class.

The following figure illustrates the event in C#.



## Declare an Event

An event can be declared in two steps:

1. Declare a delegate.
2. Declare a variable of the delegate with event keyword.

The following example shows how to declare an event in publisher class.

Part 3:

// Inheritance in C# is a mechanism of consuming the members that are defined in one class from another class by establishing a parent/child relationship between the classes.

Class A => Parent/ Base/ Superclass (all are meaning the same; you can use any term)  
Class B => Child/ Derived/ Subclass (all are meaning the same; you can use any term)

//code re-usability

Let us learn those 6 important Rules one by one.

Rule 1: parent classes constructor must be accessible to the child class. Constructor default value is private. The reason is when the Parent class constructor is called, then only the parent class members will be initialized,

Rule 2: the child class can access the parent class members but the parent classes can never access any members that are purely defined in the child class.

System is parent class.

In C# we don’t have support for multiple inheritances through classes,

(Single, Multilevel and Hierarchical supported) (i.e. Multiple and Hybrid are not supported).

Constructor : base(10)

B(int num) : base(num)

Types of inheritance :

1. Single Inheritance A-> B
2. Multi-Level Inheritance A -> B -> C
3. Hierarchical Inheritance A -. B A-> C A-> D
4. Hybrid Inheritance - Hybrid Inheritance is the inheritance that is the combination of any Single, Hierarchical, and Multilevel inheritances. For a better understanding, please have a look at the below image.
5. Multiple Inheritance A-> B, C

InterFace:

**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**"Saving Account:"**)**;

IBankAccount savingAccount = new SavingAccount**()**;

savingAccount.DepositAmount**(**2000**)**;

savingAccount.DepositAmount**(**1000**)**;

savingAccount.WithdrawAmount**(**1500**)**;

savingAccount.WithdrawAmount**(**5000**)**;

Console.WriteLine**(**$"Saving Account Balanace: {savingAccount.CheckBalance()}"**)**;

Console.WriteLine**(**"\nCurrent Account:"**)**;

IBankAccount currentAccount = new CurrentAccount**()**;

currentAccount.DepositAmount**(**500**)**;

currentAccount.DepositAmount**(**1500**)**;

currentAccount.WithdrawAmount**(**2600**)**;

currentAccount.WithdrawAmount**(**1000**)**;

Console.WriteLine**(**$"Current Account Balanace: {currentAccount.CheckBalance()}"**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**public** **class** CurrentAccount : IBankAccount

**{**

**private** **decimal** Balance = 0;

**public** **bool** DepositAmount**(decimal** Amount**)**

**{**

Balance = Balance + Amount;

Console.WriteLine**(**$"You have Deposited: {Amount}"**)**;

Console.WriteLine**(**$"Your Account Balance: {Balance}"**)**;

**return** **true**;

**}**

**public** **bool** WithdrawAmount**(decimal** Amount**)**

**{**

**if** **(**Balance **<** Amount**)**

**{**

Console.WriteLine**(**"You have Insufficient balance!"**)**;

**return** **false**;

**}**

**else**

**{**

Balance = Balance - Amount;

Console.WriteLine**(**$"You have Successfully Withdraw: {Amount}"**)**;

Console.WriteLine**(**$"Your Account Balance: {Balance}"**)**;

**return** **true**;

**}**

**}**

**public** **decimal** CheckBalance**()**

**{**

**return** Balance;

**}**

**}**

**public** **class** SavingAccount : IBankAccount

**{**

**private** **decimal** Balance = 0;

**private** **readonly** **decimal** PerDayWithdrawLimit = 10000;

**private** **decimal** TodayWithdrawal = 0;

**public** **bool** DepositAmount**(decimal** Amount**)**

**{**

Balance = Balance + Amount;

Console.WriteLine**(**$"You have Deposited: {Amount}"**)**;

Console.WriteLine**(**$"Your Account Balance: {Balance}"**)**;

**return** **true**;

**}**

//Maximum Withdrawal Per Day: 10000

**public** **bool** WithdrawAmount**(decimal** Amount**)**

**{**

**if** **(**Balance **<** Amount**)**

**{**

Console.WriteLine**(**"You have Insufficient balance!"**)**;

**return** **false**;

**}**

**else** **if** **(**TodayWithdrawal + Amount **>** PerDayWithdrawLimit**)**

**{**

Console.WriteLine**(**"Withdrawal attempt failed!"**)**;

**return** **false**;

**}**

**else**

**{**

Balance = Balance - Amount;

TodayWithdrawal = TodayWithdrawal + Amount;

Console.WriteLine**(**$"You have Successfully Withdraw: {Amount}"**)**;

Console.WriteLine**(**$"Your Account Balance: {Balance}"**)**;

**return** **true**;

**}**

**}**

**public** **decimal** CheckBalance**()**

**{**

**return** Balance;

**}**

**}**

**interface** IBankAccount

**{**

**bool** DepositAmount**(decimal** amount**)**;

**bool** WithdrawAmount**(decimal** amount**)**;

**decimal** CheckBalance**()**;

**}**

##### ****Example to Understand Abstract Class and Abstract Methods in C#:****

**Cannot create instance of abstract class**

Because it is not a fully implemented class as its abstract methods cannot be executed.**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

AbsChild absChild = new AbsChild**()**;

absChild.Add**(**10, 5**)**;

absChild.Sub**(**10, 5**)**;

absChild.Mul**(**10, 5**)**;

absChild.Div**(**10, 2**)**;

Console.ReadKey**()**;

**}**

**}**

**public** **abstract** **class** AbsParent

**{**

**public** **void** Add**(int** x, **int** y**)**

**{**

Console.WriteLine**(**$"Addition of {x} and {y} is : {x + y}"**)**;

**}**

**public** **void** Sub**(int** x, **int** y**)**

**{**

Console.WriteLine**(**$"Subtraction of {x} and {y} is : {x - y}"**)**;

**}**

**public** **abstract** **void** Mul**(int** x, **int** y**)**;

**public** **abstract** **void** Div**(int** x, **int** y**)**;

**}**

**public** **class** AbsChild : AbsParent

**{**

**public** **override** **void** Mul**(int** x, **int** y**)**

**{**

Console.WriteLine**(**$"Multiplication of {x} and {y} is : {x \* y}"**)**;

**}**

**public** **override** **void** Div**(int** x, **int** y**)**

**{**

Console.WriteLine**(**$"Division of {x} and {y} is : {x / y}"**)**;

**}**

**}**

Instance & Reference

1. A method that does not have a body is called an abstract method and the class that is declared by using the keyword abstract is called an abstract class. If a class contains an abstract method, then it must be declared as abstract.
2. An abstract class can contain both abstract and non-abstract methods. If a child class of an abstract class wants to consume any non-abstract methods of its parent, should implement all abstract methods of its parent.
3. An abstract class is never usable to itself because we cannot create the object of an abstract class. The members of an abstract class can be consumed only by the child class of the abstract class.

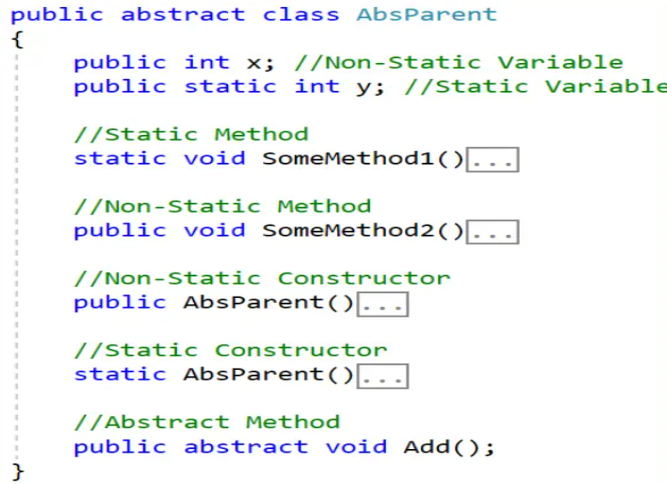
##### ****What is an Abstract Class in C#?****

A class that is declared by using the keyword abstract is called an abstract class. An abstract class is a partially implemented class used for implementing some of the methods of an object which are common for all next-level subclasses i.e. all child classes and the remaining abstract methods to be implemented by the child classes. So, it contains both abstract methods and concrete methods (non-abstract methods) including variables, properties, and indexers.

##### ****When to use the Abstract Method in C#?****

Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in a different manner. You can also do the same thing using an interface also. But if we are using an abstract class means we can provide some common functionality that is going to be the same for all the child classes and this is not possible using the interface.

We can define all static and non-static members including properties, fields, indexes, and also abstract methods.



**polymorphism**

The word polymorphism is derived from two Greek words: poly and morphs.

Polymorphism is a concept by which we can perform a single task in different ways.

##### ****Types of Polymorphism in C#:****

1. ****Static Polymorphism / Compile-Time Polymorphism / Early Binding -> Method overloading , Operator, Method hiding****
2. ****Dynamic Polymorphism / Run-Time Polymorphism / Late Binding -> Method Overriding****

****Runtime polymorphism :**** In dynamic polymorphism, the behavior of a method is decided at runtime, therefore, the CLR (Common Language Runtime) binds the method call with method definition/body at runtime and invokes the relevant method during runtime when the method is called.

**Same method name and return type different :**

Still, you have doubts, the return types are different. See, the return types come into the picture at the end of the method execution. But here, the confusion is not at the end of the method execution, but the confusion is about where to start, and which method to invoke. So, the compiler does not have any clarity to start the method execution, and talking about the end of the method execution does not make any sense. So, this is the reason why return type is never taken into consideration while defining method overloading in C#.

//Method Hiding:

**namespace** *MethodHiding*

**{**

**public** **class** Parent

**{**

**public** **virtual** **void** Show**()**

**{**

Console.WriteLine**(**"Parent Class Show Method"**)**;

**}**

**public** **void** Display**()**

**{**

Console.WriteLine**(**"Parent Class Display Method"**)**;

**}**

**}**

**public** **class** Child : Parent

**{**

//Method Overriding

**public** **override** **void** Show**()**

**{**

Console.WriteLine**(**"Child Class Show Method"**)**;

**}**

//Method Hiding/Shadowing

**public** new **void** Display**()**

**{**

Console.WriteLine**(**"Child Class Display Method"**)**;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

Child obj = new Child**()**;

obj.Show**()**;

obj.Display**()**;

Console.ReadKey**()**;

**}**

**}**

**}**

Operator :

**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

Complex c1 = new Complex**(**3, 7**)**;

c1.Display**()**;

Complex c2 = new Complex**(**5, 2**)**;

c2.Display**()**;

Complex c3 = c1 + c2;

c3.Display**()**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Complex

**{**

**private** **int** real;

**private** **int** img;

**public** Complex**(int** r = 0, **int** i = 0**)**

**{**

real = r;

img = i;

**}**

**public** **static** Complex **operator** +**(**Complex c1, Complex c2**)**

**{**

Complex temp = new Complex**()**;

temp.real = c1.real + c2.real;

temp.img = c1.img + c2.img;

**return** temp;

**}**

**public** **void** Display**()**

**{**

Console.WriteLine**(**$"{real} + i{img}"**)**;

**}**

**}**;

**public** **static** Complex Add**(**Complex c1, Complex c2**)**

**{**

Complex temp = new Complex**()**;

temp.real = c1.real + c2.real;

temp.img = c1.img + c2.img;

**return** temp;

**}**

Part2 :

 Classes and Objects

- Static, Readonly, Const

- Access Modifier

- ref, out, this // in

- Type conversion (Implicit, explicit, parse, tryparse, conversion methods)

- Constructor, Static, Overload etc C:\Users\LENOVO\Desktop\Besant\_technologies\code\Programs\.netprograms\08. Properties & Indexers\01. Creating Properties

- Inheritance, (Types of inheritance)

hiding, Overriding

Encapsulation

 Polymorphism

 Data Abstraction

- Abstract classes, Interfaces

Destructors

 Structures, Enumerations

 Boxing & Unboxing

//Program : this keyword:

//Program : Extenstion Methods:

public static class ExtensionMethod

{

public static string IsGreaterThan(this string i, string value)

{

if(i.Contains("fail"))

{

return "betterluck " + value;

}

else {

return "Congrats " + value;

}

}

}

// Program : Is & AS

/\*// creating and initializing object array

object[] o = new object[5];

o[0] = new Y();

o[1] = new Z();

o[2] = "Hello";

o[3] = 4759.0;

o[4] = null;

for (int q = 0; q < o.Length; ++q)

{

// using as operator

string str1 = o[q] as string;

bool str2 = o[q] is string;

Console.Write("{0}:", q);

// checking for the result

if (str1 != null && str2)

{

Console.WriteLine("'" + str1 + "'");

}

else

{

Console.WriteLine("Is is not a string");

}

}

//Program : Ref & Out

/\*

Ref Keyword

The ref keyword passes arguments by reference. It means any changes made to this argument in the method will be reflected in that variable when control returns to the calling method.\*/

public static string GetNextName(ref int id)

{

string returnText = "Next-" + id.ToString();

id += 1;

return returnText;

}

static void Main(string[] args)

{

int i = 1;

Console.WriteLine("Previous value of integer i:" + i.ToString());

string test = GetNextName(ref i);

Console.WriteLine("Current value of integer i:" + i.ToString());

}

/\*

Ref Out

The parameter or argument must be initialized first before it is passed to ref. [It is not compulsory to initialize a parameter or argument before it is passed to an out.]

It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method. [A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method.]

/\*

Abstract classes are used to define a common set of behaviors or properties that derived classes should have.

Abstraction in C# is the process to hide the internal details and show only the functionality. The abstract modifier indicates the incomplete implementation. The keyword abstract is used before the class or method to declare the class or method as abstract

Abstract Method: A method that is declared abstract, has no “body” and is declared inside the abstract class only. An abstract method must be implemented in all non-abstract classes using the override keyword.

an abstract class is an incomplete class or a special class we can’t be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules that the derived classes must implement when they inherit an abstract class. We can use an abstract class as a base class and all derived classes must implement abstract definitions.

An abstract class cannot be inherited by structures.

It can contain constructors or destructors.

It can implement functions with non-Abstract methods.

It cannot support multiple inheritances.

It can’t be static.

\*/

An interface can contain

Abstract methods

Properties

Indexes

Events

An interface cannot contain

Non-abstract functions

Data fields

Constructors

Destructors

/\*

What are Destructors in C#?

According to MSDN, Destructors which are also called Finalizers in C# are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.

tilde (~) symbol

//Program: Destructor:

using System;

namespace DestructorExample

{

class DestructorDemo

{

public DestructorDemo()

{

Console.WriteLine("Constructor Object Created");

}

~DestructorDemo()

{

string type = GetType().Name;

Console.WriteLine($"Object {type} is Destroyed");

}

}

class Program

{

static void Main(string[] args)

{

DestructorDemo obj1 = new DestructorDemo();

DestructorDemo obj2 = new DestructorDemo();

//Making obj1 and obj2 ready for Garbage Collection

obj1 = null;

obj2 = null;

GC.Collect();

Console.ReadKey();

}

}

}

// Program : Dispose

using System;

namespace DestructorDemo

{

public class UmmanagedResource : IDisposable

{

#region IDisposable Support

private bool disposedValue = false; // To detect redundant calls

protected virtual void Dispose(bool disposing)

{

if (!disposedValue)

{

if (disposing)

{

//Write Code Here to Destroy the Managed Resources

Console.WriteLine("Managed Resources Destroyed by Dispose Method");

}

//Write Code Here to Destroy the Umanaged Resources

Console.WriteLine("Unmanaged Resources Destroyed by Dispose Method");

disposedValue = true;

}

else

{

Console.WriteLine("Resources are Already Destroyed by Dispose Method");

}

}

~UmmanagedResource()

{

//Write Code here to Destroy the Object

Console.WriteLine("Request Comes to Destructor to Destroy the Resources");

Dispose(false);

}

// This code added to correctly implement the disposable pattern.

public void Dispose()

{

//Write Code here to Destroy the Object

Console.WriteLine("Request Comes to Dispose Method to Destroy the Resources");

Dispose(true);

GC.SuppressFinalize(this);

}

#endregion

}

class Program

{

static void Main()

{

UmmanagedResource resource = null;

try

{

resource = new UmmanagedResource();

Console.WriteLine("Using Resources");

}

finally

{

if (resource != null)

{

Console.WriteLine("Calling Dispose Method to Destroy Resources");

resource.Dispose();

}

}

//Trying to Call the Dispose Method again

Console.WriteLine();

Console.WriteLine("Trying to Call the Dispose Method Again To Destroy Resources");

resource.Dispose();

Console.ReadKey();

}

}

}

\*/

///////////////////////////////////////////////////////////////////

/\* C# Basics Examples:

Step1 : Open visual studio

Step2 : Create console application

Step3 : Copy & Paste below programs one by one and execute the program.

Program 1: Hello World

======================

\*/

﻿class Sample

{

static void Main()

{

System.Console.WriteLine("Hello");

System.Console.ReadKey();

}

}

//Progrm 2: Console Class program

//================================

﻿class Sample

{

static void Main()

{

//Print messages line-by-line

System.Console.WriteLine("Welcome");

System.Console.WriteLine("to");

System.Console.WriteLine("C# Programming");

//Print messages side-by-side

System.Console.Write("Welcome ");

System.Console.Write("to ");

System.Console.Write("C# Programming");

//Wait for pressing some key on the keyboard

System.Console.ReadKey();

//clear the screen

System.Console.Clear();

System.Console.WriteLine("Thank you.");

System.Console.ReadKey();

}

}

/Program 3: Variables Example

//=============================

﻿class Sample

{

static void Main()

{

//declare the variable

string studentName = "Scott";

int age = 20;

//print value

System.Console.Write("Hey ");

System.Console.Write(studentName);

System.Console.Write(" , your age is ");

System.Console.Write(age);

//wait for pressing some key on keyboard

System.Console.ReadKey();

}

}

// Program 4: Primitive Types Example

//=====================================

﻿class Sample

{

static void Main()

{

sbyte a = sbyte.MaxValue;

sbyte b = 100;

sbyte c = default(sbyte);

byte d = byte.MaxValue;

short e = short.MaxValue;

ushort f = ushort.MaxValue;

int g = int.MaxValue;

uint h = uint.MaxValue;

long i = long.MaxValue;

ulong j = ulong.MaxValue;

float k = float.MaxValue;

double l = double.MaxValue;

decimal m = decimal.MaxValue;

char n = 'A';

string o = "abc";

bool p = true;

System.Console.WriteLine(a);

System.Console.WriteLine(b);

System.Console.WriteLine(c);

System.Console.WriteLine(d);

System.Console.WriteLine(e);

System.Console.WriteLine(f);

System.Console.WriteLine(g);

System.Console.WriteLine(h);

System.Console.WriteLine(i);

System.Console.WriteLine(j);

System.Console.WriteLine(k);

System.Console.WriteLine(l);

System.Console.WriteLine(m);

System.Console.WriteLine(n);

System.Console.WriteLine(o);

System.Console.WriteLine(p);

System.Console.ReadKey();

}

}

//Program 5 : Operators:

//=======================

﻿class Program

{

static void Main()

{

//Arithmetical Operators

decimal a = 10M;

decimal b = 3M;

decimal c = a + b; //Output: 13

decimal d = a - b; //Output: 7

decimal e = a \* b; //Output: 30

decimal f = a / b; //Output: 3.3333333

decimal g = a % b; //Output: 1

System.Console.WriteLine(c);

System.Console.WriteLine(d);

System.Console.WriteLine(e);

System.Console.WriteLine(f);

System.Console.WriteLine(g);

//Assignment Operators

a += 20M;

System.Console.WriteLine(a); //Output: 30

a -= 20M;

System.Console.WriteLine(a); //Output: 10

a \*= 3M;

System.Console.WriteLine(a); //Output: 30

a /= 3M;

System.Console.WriteLine(a); //Output: 10

a %= 3M;

System.Console.WriteLine(a); //Output: 1

//Increment / Decrement Operators

a = 10M;

System.Console.WriteLine();

System.Console.WriteLine(++a); //Output: 11

System.Console.WriteLine(a++); //Output: 11

System.Console.WriteLine(a); //Output: 12

System.Console.WriteLine(--a); //Output: 11

System.Console.WriteLine(a--); //Output: 11

System.Console.WriteLine(a); //Output: 10

//Comparison Operators

System.Console.WriteLine();

bool b1 = a == 10;

System.Console.WriteLine(b1); //Output: true

bool b2 = a != 10;

System.Console.WriteLine(b2); //Output: false

bool b3 = a < 10;

System.Console.WriteLine(b3); //Output: false

bool b4 = a > 10;

System.Console.WriteLine(b4); //Output: false

bool b5 = a <= 10;

System.Console.WriteLine(b5); //Output: true

bool b6 = a >= 10;

System.Console.WriteLine(b6); //Output: true

//Logical Operators

System.Console.WriteLine();

bool b7 = a == 10 & b == 10;

System.Console.WriteLine(b7); //Output: false

bool b8 = a == 10 && b == 10;

System.Console.WriteLine(b8); //Output: false

bool b9 = a == 10 | b == 10;

System.Console.WriteLine(b9); //Output: true

bool b10 = a == 10 || b == 10;

System.Console.WriteLine(b10); //Output: true

bool b11 = !(a == 10);

System.Console.WriteLine(b11); //Output: false

bool b12 = a == 10 ^ b == 10;

System.Console.WriteLine(b12); //Output: true

//concatenation operator

string name = "Scott";

int age = 20;

string message = "Hey " + name + ", your age is " + age + ".";

System.Console.WriteLine(message);

//ternary operator

string title = (age < 13) ? "Child" : (age >= 13 && age <= 19) ? "Teenager" : "Adult";

System.Console.WriteLine(title);

//operator precedence

double z = 10 + 4 \* 30 / 10;

System.Console.WriteLine(z); //Output: 22

System.Console.ReadKey();

}

}

//Program 6 : IF, Else, Else IF, Nested IF

//===========================================

﻿class Program

{

static void Main()

{

//declare variable to store marks of the student

int marks = 45;

char gradeLetter;

//>=85 O

//>=60 && <85 A

//>=50 && < 60 B

//>=35 && < 50 C

//else F

if (marks >= 35)

{

if (marks >= 85)

{

gradeLetter = 'O';

}

else if (marks >= 60 && marks < 85)

{

gradeLetter = 'A';

}

else if (marks >= 50 && marks < 60)

{

gradeLetter = 'B';

}

else

{

gradeLetter = 'C';

}

}

else

{

gradeLetter = 'F';

}

System.Console.WriteLine(gradeLetter);

System.Console.ReadKey();

}

}

//Program 7 : Switch Case

//=======================

﻿class Program

{

static void Main()

{

//variable to store grade letter of student

char gradeLetter = 'A';

//find out description based on gradeLetter

string gradeDescription;

switch(gradeLetter)

{

case 'O': gradeDescription = "Outstanding"; break;

case 'A': gradeDescription = "Excellent"; break;

case 'B': gradeDescription = "Good"; break;

case 'C': gradeDescription = "Average"; break;

case 'F': gradeDescription = "Fail"; break;

default: gradeDescription = "None"; break;

}

System.Console.WriteLine(gradeDescription);

System.Console.ReadKey();

}

}

// Program 8 : While & Do while

//=============================

﻿class Program

{

static void Main()

{

#region While

//1 to 10

int i = 1;

while ( i <= 10)

{

System.Console.Write(i + " ");

i++;

}

System.Console.WriteLine();

#endregion

//1 to 10

int i = 1;

do

{

System.Console.Write(i + " ");

i++;

} while (i <= 10);

System.Console.WriteLine();

//0 to 9

i = 0;

do

{

System.Console.Write(i + " ");

i++;

} while (i < 10);

System.Console.WriteLine();

//9 to 0

i = 9;

do

{

System.Console.Write(i + " ");

i--;

} while (i >= 0);

System.Console.ReadKey();

}

}

// Program 8 : For Loop

//======================

﻿class Program

{

static void Main()

{

//1 to 10

for (int i = 1; i <= 10; i++)

{

System.Console.Write(i + " ");

}

System.Console.WriteLine();

//0 to 9

for (int i = 0; i < 10; i++)

{

System.Console.Write(i + " ");

}

System.Console.WriteLine();

//9 to 0

int j = 9;

for (; j >= 0; j--)

{

System.Console.Write(j + " ");

}

System.Console.ReadKey();

}

}

// Program 9 : Break

//==================

﻿class Program

{

static void Main()

{

//for loop: 0 to 9

for (int i = 0; i < 10; i++)

{

if (i == 5)

{

break;

}

System.Console.WriteLine(i);

}

System.Console.ReadKey();

}

}

// Program 10 : Continue

//======================

﻿class Program

{

static void Main()

{

//loop

for (int i = 1; i <= 10; i++)

{

if (i == 6)

{

continue;

}

System.Console.WriteLine(i);

}

System.Console.ReadKey();

}

}

// Program 11 : Nested Loop

//==========================

﻿class Program

{

static void Main()

{

for (int i = 1; i <= 5; i++)

{

for (int j = 1; j <= i; j++)

{

System.Console.Write(j);

System.Console.Write(" ");

}

System.Console.WriteLine();

}

System.Console.ReadKey();

}

}

// Program 12 : GOTO

//===================

﻿class Program

{

static void Main()

{

int i = 1;

System.Console.WriteLine("USA");

System.Console.WriteLine("UK");

System.Console.WriteLine("India");

mylabel:

System.Console.WriteLine("France");

System.Console.WriteLine("Italy");

System.Console.WriteLine("Iran");

i++;

if (i <=5)

{

goto mylabel;

}

System.Console.WriteLine("Nepal");

System.Console.WriteLine("Dubai");

System.Console.ReadKey();

}

}