**Class**

**Class** (sinf) - bu obyektga yo'naltirilgan dasturlash (OOP) ning asosiy tarkibiy qismi bo'lib, unda obyektlar yaratish uchun zarur bo'lgan xususiyatlar va metodlar to'plami saqlanadi. Sinf - bu obyektning qanday xususiyatlarga ega bo‘lishini va qanday xatti-harakatlarni amalga oshirishini aniqlaydigan shablon yoki qolip.

Class ning tarifi

**Class** - bir xil turdagi obyektlarni yaratish uchun ishlatiladigan kod tuzilmasi. Unda obyektning barcha xususiyatlari (fields va properties) va imkoniyatlari (metods) e’lon qilinadi. Sinf yordamida dasturchi real hayotdagi narsalarni dasturiy ta’minotda ifodalaydi, masalan, avtomobil, talaba yoki mahsulot.

Misol:

public class Avtomobil

{

public string Model { get; set; }

public int Yili { get; set; }

public void Yurish()

{

Console.WriteLine($"{Model} yurmoqda.");

}

}

Bu yerda Avtomobil sinfi avtomobilning Model va Yili kabi xususiyatlarini belgilaydi va Yurish nomli metodni o'z ichiga oladi.

Class ning maqsadi

**Obyekt yaratish uchun shablon**: Sinf yordamida bir xil turdagi bir nechta obyektlarni yaratish mumkin. Masalan, Avtomobil sinfi yordamida turli modellarga ega bo'lgan avtomobillar yaratish mumkin.

**Kodning qayta ishlatilishi**: Sinflar kodni qayta ishlatish imkoniyatini beradi. Bir marta sinfni yaratib, undan bir nechta obyekt yaratish mumkin.

**OOP tamoyillarini amalga oshirish**: Sinflar obyektga yo'naltirilgan dasturlash tamoyillarini (Encapsulation, Inheritance, Polymorphism va Abstraction) qo‘llash imkonini beradi.

Obyekt va Class o‘rtasidagi farq

**Class** - bu shablon yoki qolip, lekin u o‘zi mustaqil ishlatilmaydi.

**Obyekt** - sinfdan yaratilgan aniq nusxa bo‘lib, u ishlatiladigan ma'lumotlar va metodlarni o‘zida saqlaydi. Masalan, Avtomobil sinfi mavjud bo‘lishi mumkin, lekin avto1 nomli obyekt aynan shu sinfdan yaratilgan konkret mashina.

Misol:

public class Avtomobil

{

public string Model { get; set; }

public int Yili { get; set; }

}

Avtomobil avto1 = new Avtomobil(); // Bu obyekt

avto1.Model = "Toyota";

avto1.Yili = 2020;

Class ning asosiy tarkibiy qismlari

**Fields**: Ma'lumotlarni bevosita saqlash uchun ishlatiladigan o‘zgaruvchilar.

**Properties**: Field larni o‘qish va yozish uchun boshqaruv elementlari.

**Methods**: Sinfning bajarishi mumkin bo‘lgan funksional imkoniyatlarini ifodalovchi funksiyalar.

**Constructors**: Sinf obyekti yaratishda ishga tushadigan maxsus metodlar bo‘lib, obyektni boshlang‘ich qiymatlar bilan to‘ldiradi.

Class va Struct farqi

|  |  |  |
| --- | --- | --- |
| Xususiyat | Class | Struct |
| **Meros olish** | Meros olishni qo'llab-quvvatlaydi | Meros olishni qo'llab-quvvatlamaydi |
| **Qayerda saqlanadi** | Heap (Heap xotirada saqlanadi) | Stack (Stack xotirada saqlanadi) |
| **Ma'lumot turi** | Referens tur (reference type) | Qiymat turi (value type) |
| **Null qiymat** | Null qiymat qabul qilishi mumkin | Null qiymat qabul qilmaydi |
| **Ishlatilish maqsadi** | Murakkab ob'ektlar uchun | Oddiy va kichik strukturalar uchun |

Xulosa

**Class** - obyektlar yaratish uchun shablon bo'lib, dasturda real hayotdagi obyektlarni modellashtirish uchun ishlatiladi.

**Maqsadi** - obyektlarni yaratish, ularning xususiyatlari va xatti-harakatlarini aniqlash, va kodni qayta ishlash imkoniyatini berish.

4o

class Car

{

string model;

string color;

int year;

static void Main(string[] args)

{

Car Ford = new Car();

Ford.model = "Mustang";

Ford.color = "red";

Ford.year = 1969;

Car Opel = new Car();

Opel.model = "Astra";

Opel.color = "white";

Opel.year = 2005;

Console.WriteLine(Ford.model);

Console.WriteLine(Opel.model);

}

}

**Struct**

C# tilida **struct** (struktur) - qiymat turlarini ifodalash uchun ishlatiladigan yengilroq tuzilma bo'lib, oddiy va kichik obyektlar yaratishda foydalaniladi. **Struct** yordamida kichik ma'lumotlar to'plamlarini to'liq obyekt yaratmasdan ifodalash mumkin.

Struct ning tarifi

**Struct** - qiymat turi bo'lib, unda xususiyatlar (properties), o'zgaruvchilar (fields), va metodlar bo'lishi mumkin. **Struct** ning o‘ziga xos xususiyati shundaki, u **stack** xotirada saqlanadi va o‘zi yaratgan nusxalar **qiymat turi** sifatida ishlaydi. Bu uning **class** dan farqlaridan biridir, chunki sinflar **heap** xotirada saqlanadi va **referens turi** hisoblanadi.

Struct yaratish misoli:

public struct Nuqta

{

public int X { get; set; }

public int Y { get; set; }

public void KoordinatalarniChopEt()

{

Console.WriteLine($"Nuqta koordinatalari: ({X}, {Y})");

}

}

Yuqoridagi misolda Nuqta nomli struct e'lon qilingan bo‘lib, u nuqtaning X va Y koordinatalarini saqlaydi va chop etish uchun metodga ega.

Struct ning maqsadi

**Yengil obyektlarni yaratish**: Struct yengil vaznli tuzilmalarni yaratish uchun mo‘ljallangan. Masalan, kichik koordinata nuqtalari, ranglar yoki o'lchamlar kabi oddiy ma'lumotlarni ifodalashda qo‘llaniladi.

**Qiymat turi sifatida ishlatish**: Struct qiymat turi bo‘lgani sababli, u nusxa ko‘chirishda o‘zining asl qiymatini boshqa o‘zgaruvchiga beradi. Bu joylashuv xotirasi va ishlov berish jarayonlarini optimallashtirish imkonini beradi.

**Stack xotirada saqlash**: Struct stack xotirada saqlanadi, bu esa foydalanish jarayonida xotira boshqaruvini soddalashtiradi va odatda **heap** dan ko‘ra tezroq ishlashga olib keladi.

Struct ning Class dan farqlari

|  |  |  |
| --- | --- | --- |
| Xususiyat | Struct | Class |
| **Xotirada saqlanish** | Stack (qiymat turi) | Heap (referens turi) |
| **Meros olish** | Meros ola olmaydi | Meros olish imkoniyati mavjud |
| **Null qiymat** | Null qiymat qabul qilmaydi | Null qiymat qabul qilishi mumkin |
| **Obyektlar o‘rtasidagi munosabat** | Nusxalar qiymatini beradi | Referensni beradi |
| **Qachon ishlatiladi** | Yengil vaznli obyektlar uchun | Murakkab va katta obyektlar uchun |

Structdan foydalanish misoli

Quyida Nuqta structidan foydalanish misoli keltirilgan:

Nuqta n1 = new Nuqta { X = 3, Y = 4 };

Nuqta n2 = n1; // n2 n1 ning nusxasini oldi (qiymat turi sifatida)

n2.X = 10;

Console.WriteLine($"n1.X: {n1.X}, n1.Y: {n1.Y}"); // n1.X: 3

Console.WriteLine($"n2.X: {n2.X}, n2.Y: {n2.Y}"); // n2.X: 10

Bu misolda n2 o‘ziga n1 ning qiymat nusxasini oladi, shuning uchun n1 va n2 mustaqil qiymatlarga ega bo‘lib qoladi.

Struct dan foydalanish qachon maqsadga muvofiq

Kichik, yengil vaznli ma'lumotlar tuzilmalarini yaratishda.

Qo‘shimcha meros olish va polymorphism talab qilinmasa.

Obyektlar soni ko‘p va xotira ishlatilishini optimallashtirish zarur bo‘lsa.

Xulosa

**Struct** - kichik va sodda obyektlar uchun ishlatiladigan qiymat turi bo‘lib, xotirada optimallashtirilgan saqlash imkonini beradi.

**Maqsadi** - engil ma'lumot tuzilmalarini qiymat turi sifatida ishlatib, xotirani tejash va stack xotira asosida ishlashni ta’minlash.

Constructor

**Constructor** (konstruktor) - bu **class** yoki **struct** da obyekt yaratilganda ishga tushadigan maxsus metod. Konstruktorning asosiy vazifasi - yangi yaratilgan obyekt uchun boshlang'ich qiymatlarni belgilash va obyektni ishga tayyorlash.

Constructorning asosiy xususiyatlari:

**Class yoki Struct bilan bir xil nomda bo'ladi**.

**Qaytish turi (return type)** bo'lmaydi.

**Obyekt yaratishda avtomatik ravishda** ishga tushadi.

Constructorning tarifi va turi

C# da konstruktorlar bir nechta turga ega bo'lib, ular quyidagilar:

1. Oddiy (default) konstruktor

Oddiy konstruktor parametr qabul qilmaydi va obyekt yaratilganda default (standart) qiymatlarni o‘rnatadi.

Mashina

{

public string Model;

public int Yili;

// Oddiy konstruktor

public Mashina()

{

Model = "Noma'lum";

Yili = 2023;

}

}

Bu misolda Mashina sinfining oddiy konstruktori obyekt yaratishda Model va Yili uchun standart qiymatlarni o‘rnatadi.

2. Parametrli konstruktor

Parametrli konstruktor obyekt yaratilganda qiymatlarni parametrlar orqali belgilash imkonini beradi.

public class Mashina

{

public string Model;

public int Yili;

// Parametrli konstruktor

public Mashina(string model, int yil)

{

Model = model;

Yili = yil;

}

}

Bu misolda Mashina sinfi uchun parametrli konstruktor ishlatilgan bo'lib, obyekt yaratishda Model va Yili ni belgilash mumkin.

3. Statik konstruktor

Statik konstruktor **static** xususiyatlarni ishga tushirish uchun ishlatiladi. U faqat bir marta ishlaydi va obyekt emas, balki sinf yuklanayotganda (loading) avtomatik ishga tushadi.

public class Config

{

public static string APIKey;

// Statik konstruktor

static Config()

{

APIKey = "12345";

}

}

Bu misolda Config sinfida statik konstruktor ishlatilgan bo‘lib, APIKey ni bir marta o‘rnatadi.

Constructorning maqsadi

**Obyektni boshlang‘ich qiymat bilan ta'minlash**: Konstruktor yordamida obyekt yaratilganda zarur boshlang'ich qiymatlar o‘rnatiladi.

**Obyektni foydalanishga tayyorlash**: Konstruktor obyekt uchun zarur resurslarni yaratadi yoki tayyorlaydi.

**Parametrlar orqali moslashtirish**: Parametrli konstruktor orqali obyekt yaratishda kerakli qiymatlarni belgilashga imkon beradi.

Constructor va Method o‘rtasidagi farq

| **Xususiyat** | **Constructor** | **Method** |
| --- | --- | --- |
| **Qaytish turi** | Yo‘q | Qaytish turi bor |
| **Class nomi bilan** | Class yoki struct nomi bilan bir xil | Class nomidan farq qilishi mumkin |
| **Ishga tushishi** | Obyekt yaratishda avtomatik ishga tushadi | Faqat chaqirilganda ishlaydi |
| **Maqsadi** | Obyektni boshlang‘ich qiymatlar bilan ta'minlash | Funktsiyalarni bajarish uchun |

Misol - Obyekt yaratishda konstruktorlardan foydalanish

Quyidagi misolda Mashina sinfidan foydalanishda konstruktorlarning qanday ishlashini ko‘rish mumkin:

public class Mashina

{

public string Model;

public int Yili;

// Oddiy konstruktor

public Mashina()

{

Model = "Noma'lum";

Yili = 2023;

}

// Parametrli konstruktor

public Mashina(string model, int yil)

{

Model = model;

Yili = yil;

}

}

// Obyekt yaratish

Mashina avto1 = new Mashina(); // Oddiy konstruktor ishga tushadi

Mashina avto2 = new Mashina("Toyota", 2020); // Parametrli konstruktor ishga tushadi

Xulosa

**Constructor** obyekt yaratishda ishga tushadigan maxsus metod bo‘lib, obyektning boshlang‘ich qiymatlarini belgilash uchun ishlatiladi.

**Maqsadi** - obyektni ishga tayyorlash va zarur boshlang‘ich qiymatlarni o‘rnatish.

Funksiya

C# tilida **Method** (metod) — bu biror **class** yoki **struct** ichida joylashgan, ma'lum bir vazifani bajaradigan funksiyadir. Metodlar obyektlarga xatti-harakatlarni bajarish imkonini beradi. Boshqacha aytganda, metodlar — bu sinflar va strukturalarning a'zolari bo'lib, ular obyektlar bilan ishlashda yoki umumiy funksiyalarni bajarishda ishlatiladi.

Methodning asosiy xususiyatlari:

**Qaytish turi**: Har bir metod ma'lum bir qiymatni qaytarishi yoki hech narsa qaytarmasligi mumkin (void).

**Nom**: Har bir metodning o‘ziga xos nomi bo‘ladi.

**Parametrlar**: Metodga kirish uchun zarur bo‘lgan qiymatlar parametrlar orqali uzatiladi.

**Ishlash turi**: Metodlar kodni bajarish uchun bir qator operatsiyalarni amalga oshiradi.

Methodning umumiy sintaksisi

returnType MethodName(parameter1, parameter2, ...)

{

// Metodning ichidagi kod

return result; // Agar metod qaytish turi belgilangan bo'lsa

}

Misollar:

1. Oddiy metod (void)

void metod hech qanday qiymat qaytarmaydi, faqat biror amalni bajaradi.

public class Salomlashish

{

public void SalomBer()

{

Console.WriteLine("Salom, dunyo!");

}

}

Yuqoridagi misolda, SalomBer metodining qaytish turi void bo'lib, u faqat ekranga "Salom, dunyo!" deb chiqarish vazifasini bajaradi.

2. Parametrli metod

Metodlar parametrlar qabul qilib, ularni ishlatishlari mumkin.

public class Kalkulyator

{

public int Qo‘shish(int a, int b)

{

return a + b;

}

}

Bu misolda, Qo‘shish metodi ikki parametrni qabul qiladi (a va b), ularni qo‘shadi va natijani qaytaradi.

3. Qaytish turi bo'lgan metod

Metod qaytish turi bo‘lsa, u o‘z hisobidan qiymat qaytarishi kerak.

public class Mashina

{

public int Hisoblash(int narx, int miqdor)

{

return narx \* miqdor; // Narx va miqdorni ko'paytirib qaytaradi

}

}

Bu misolda, Hisoblash metodi ikki parametrni qabul qilib, ularni ko'paytirib natijani qaytaradi.

4. Statik metod

Statik metodlar sinfning o‘ziga tegishli bo‘lib, ularni sinf nomi orqali chaqirish mumkin. Statik metodlarga obyekt yaratish shart emas.

public class Matematika

{

public static int Kvadrat(int a)

{

return a \* a;

}

}

Yuqoridagi misolda Kvadrat metodi statik bo‘lib, uni sinf nomi orqali chaqirish mumkin:

int natija = Matematika.Kvadrat(5); // 25 ni qaytaradi

5. Overloading (Metodni ortiqcha yuklash)

C# da metodlar bir xil nomga ega bo‘lishi mumkin, lekin ularning parametr turlari yoki soni boshqacha bo‘lishi kerak.

public class Kalkulyator

{

// Qo‘shish metodining birinchi versiyasi

public int Qo‘shish(int a, int b)

{

return a + b;

}

// Qo‘shish metodining ikkinchi versiyasi

public double Qo‘shish(double a, double b)

{

return a + b;

}

}

Bu misolda, Qo‘shish metodining ikkita versiyasi mavjud: biri int turlarini qabul qiladi, ikkinchisi esa double turlarini qabul qiladi.

Methodning maqsadi:

**Xatti-harakatlarni ifodalash**: Metodlar obyektlar uchun bajarilishi kerak bo‘lgan vazifalarni ifodalaydi, masalan, ma'lumotlarni qayta ishlash, ekranga chiqarish yoki hisoblash.

**Kodning qayta ishlatilishi**: Metodlar orqali kodni qayta ishlatish mumkin, shuning uchun metodlar bir necha marta chaqirilishi mumkin.

**Abstraktsiya va encapsulation**: Metodlar yordamida kodni tashqi dunyodan yashirish va faqat kerakli interfeysni taqdim etish mumkin. Bu obyektga yo'naltirilgan dasturlash (OOP) ning muhim tamoyillaridan biridir.

Method va Constructor o‘rtasidagi farq:

|  |  |  |
| --- | --- | --- |
| Xususiyat | Method | Constructor |
| **Maqsadi** | Funksional kodni bajarish | Obyektni yaratish va boshlang‘ich qiymatlarni o‘rnatish |
| **Qaytish turi** | Bor (odatda) | Yo‘q (void) |
| **Chaqarish** | Har doim chaqirilishi mumkin | Obyekt yaratishda avtomatik chaqiriladi |
| **O‘zgartirilishi** | Parametrlar orqali moslashtirilishi mumkin | O‘zgartirilmaydi |

Xulosa

**Method** - bu sinf yoki struktura ichida joylashgan, aniq bir funksiyalarga xizmat qiladigan kod bo‘lagi bo‘lib, u parametrlar qabul qilib, qiymat qaytarishi mumkin. Metodlar obyektlar uchun xatti-harakatlarni bajarishda, kodni qayta ishlatishda va dastur tuzilishini soddalashtirishda katta rol o'ynaydi.

**Access Modifiers**

C# tilida **Access Modifiers** (kirish modifikatorlari) — bu sinf yoki uning a'zolariga (masalan, o'zgaruvchilar, metodlar, konstruktorlar) kirishni boshqaradigan modifierlardir. Kirish modifikatorlari yordamida siz qaysi sinflar yoki obyektlar metod va xususiyatlarga kirish imkoniyatiga ega bo'lishini belgilashingiz mumkin.

Acces Modifiers C# da mavjud bo'lgan asosiy **Access Modifiers** (kirish modifikatorlari) quyidagilar:

1. **public**

public modifikatori sinf yoki uning a'zolariga **hamma joydan** kirishga imkon beradi. Bu eng ochiq modifikator bo'lib, obyekt va sinfga qanday joydan bo'lishidan qat'iy nazar kirish mumkin.

public class Mashina

{

public string Model;

public void SalomBer()

{

Console.WriteLine("Salom, mashina!");

}

}

Yuqoridagi misolda Model va SalomBer metodiga sinfning barcha obyektlaridan va boshqa sinflardan kirish mumkin.

2. **private**

private modifikatori sinfning ichidagi a'zolariga faqat shu sinfdan kirishga imkon beradi. **Tashqi sinflardan** yoki obyektlardan **private** a'zolarga kirish mumkin emas. Bu modifikator orqali ma'lumotlarni yashirish (encapsulation) imkoniyatini beradi.

public class Mashina

{

private string Model;

private void SalomBer()

{

Console.WriteLine("Salom, private mashina!");

}

}

Bu yerda Model va SalomBer faqat Mashina sinfi ichida ishlatilishi mumkin, tashqi sinflardan yoki obyektlardan kirish taqiqlangan.

3. **protected**

protected modifikatori faqat o‘z sinfi va **o‘rgangan (inherit)** sinflardan kirish imkonini beradi. Boshqacha aytganda, protected faqat sinf va uning meros olgan (derivativ) sinflari uchun mavjud.

public class Mashina

{

protected string Model;

protected void SalomBer()

{

Console.WriteLine("Salom, protected mashina!");

}

}

public class SportMashina : Mashina

{

public void TaqdimEt()

{

Console.WriteLine(Model); // Modelga kirish mumkin

SalomBer(); // SalomBer() metodiga kirish mumkin

}

}

Bu misolda, SportMashina sinfi Mashina sinfidan meros olganligi sababli, Model va SalomBer metodiga kirish huquqiga ega.

4. **internal**

internal modifikatori **faqat bir xil yig‘inda (assembly) yoki kutubxonada** ishlatiladigan a'zolarni anglatadi. Agar a'zo internal bo'lsa, uni boshqa yig‘inlarda (kutubxonalarda) chaqirish mumkin emas.

internal class Mashina

{

internal string Model;

}

Yuqoridagi misolda, Mashina sinfi va Model xususiyatiga faqat shu yig‘in (assembly) ichidan kirish mumkin.

5. **protected internal**

protected internal modifikatori sinfning o'zidan, uning meros olgan sinflaridan yoki **bir xil yig‘indan** (assembly) bo‘lgan sinflardan kirishga imkon beradi.

public class Mashina

{

protected internal string Model;

}

Bu misolda, Model xususiyatiga faqat Mashina sinfi, uning meros olgan sinflari yoki shu yig‘indagi boshqa sinflar kirish imkoniyatiga ega.

6. **private protected**

private protected modifikatori faqat sinf ichida va uning meros olgan sinflaridan kirish imkonini beradi, ammo **tashqi yig‘inda** kirish taqiqlangan.

public class Mashina

{

private protected string Model;

}

Bu misolda, Model xususiyatiga faqat Mashina sinfi va uning meros olgan sinflari kirish imkoniyatiga ega, lekin boshqa yig‘indagi sinflar kirish mumkin emas.

Access Modifiers o‘rtasidagi farqlar

|  |  |  |  |
| --- | --- | --- | --- |
| Modifikator | Tashqi sinflardan kirish | Meros olish orqali kirish | Yig‘inda kirish (Assembly) |
| **public** | Mavjud | Mavjud | Mavjud |
| **private** | Yo'q | Yo'q | Yo'q |
| **protected** | Yo'q | Mavjud | Yo'q |
| **internal** | Yo'q | Yo'q | Mavjud |
| **protected internal** | Mavjud | Mavjud | Mavjud |
| **private protected** | Yo'q | Mavjud | Yo'q |

Xulosa

**Access modifiers** sinfning ichidagi a'zolarga kirishni boshqaradi va ma'lumotlarni yashirish imkonini beradi.

**public** - eng ochiq modifikator bo'lib, barcha joylardan kirishga imkon beradi.

**private** - faqat sinf ichidan kirishga imkon beradi.

**protected** - sinf va uning meros olgan sinflaridan kirishga imkon beradi.

**internal** - faqat bir xil yig‘inda (assembly) ishlatiladi.

**protected internal** - sinf, uning meros olgan sinflari va yig‘indagi boshqa sinflardan kirishga imkon beradi.

**private protected** - sinf va uning meros olgan sinflaridan kirishga imkon beradi, lekin boshqa yig‘indagi sinflar kirish ololmaydi.

OOP

C# tilidagi obyektga yo'naltirilgan dasturlash (OOP) tamoyillari to'rtta asosiy qismga bo'linadi: **Inheritance (Meros olish)**, **Encapsulation (Kapsulyatsiya)**, **Polymorphism (Polimorfizm)**, va **Abstraction (Abstraksiya)**. Bu tamoyillar kodni modullash va murakkab dasturlarni boshqarishni osonlashtiradi. Quyida ularning tariflari, maqsadlari va o'zaro farqlari bilan tanishib chiqamiz:

**1. Inheritance (Meros olish)**

**Tarif**: Meros olish orqali bir sinf boshqa sinfning xususiyatlari va metodlarini o'zida mujassam qilishi mumkin. Bu orqali mavjud sinfning xususiyatlari yangi sinfga o'rnatiladi.

**Maqsad**: Kodni qayta ishlatish va sinflar orasida o'xshashlikni saqlash. Meros olish orqali bir nechta sinflar uchun umumiy xususiyatlarni bitta asosiy sinfda to'plab, ularni yangilariga meros qilib berish mumkin.

**Farqi**: Inheritance - sinflar o'rtasidagi ierarxik munosabatlarni yaratadi, ya'ni "bu - shunday narsadir" degan munosabat (masalan, "It - Hayvondir"). Meros olishda yangi sinf (farzand sinf) eski sinfning (ota sinf) metod va xususiyatlarini avtomatik ravishda oladi.

class Vehicle // base class (parent)

{

public string brand = "Ford"; // Vehicle field

public void honk() // Vehicle method

{

Console.WriteLine("Tuut, tuut!");

}

}

class Car : Vehicle // derived class (child)

{

public string modelName = "Mustang"; // Car field

}

class Program

{

static void Main(string[] args)

{

// Create a myCar object

Car myCar = new Car();

// Call the honk() method (From the Vehicle class) on the myCar object

myCar.honk();

// Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class

Console.WriteLine(myCar.brand + " " + myCar.modelName);

}

}

**2. Encapsulation (Kapsulyatsiya)**

**Tarif**: Kapsulyatsiya - bu sinfdagi ma'lumotlarni tashqi aralashuvdan himoyalash va ularni faqat kerakli metodlar orqali boshqarishni ta'minlash usuli. Buning uchun private, public, va protected kabi kirish modifikatorlari ishlatiladi.

**Maqsad**: Ma'lumotlarni himoya qilish va ularning noto'g'ri ishlatilishini oldini olish. Bu kodni xavfsiz va barqaror qilishda yordam beradi.

**Farqi**: Encapsulation ma'lumotlarni bir joyga yig'ib, ularni sinf doirasida boshqarishni ta'minlaydi. Kapsulyatsiya ob'ekt ichidagi ma'lumotlar va funksionallikni o'rab turib, ularni tashqi kirishdan yashiradi.

class Person

{

private string name; // field

public string Name // property

{

get { return name; }

set { name = value; }

}

}

class Program

{

static void Main(string[] args)

{

Person myObj = new Person();

myObj.Name = "Liam";

Console.WriteLine(myObj.Name);

}

}

Short Hand

class Person

{

public string Name { get; set; } // property

}

class Program

{

static void Main(string[] args)

{

Person myObj = new Person();

myObj.Name = "Liam";

Console.WriteLine(myObj.Name);

}

}

**3. Polymorphism (Polimorfizm)**

**Tarif**: Polimorfizm bir xil metodning turli sinflarda turli shakllarda ishlatilishiga imkon beradi. Bu ko'p shakllilikni ifodalaydi, masalan, bir xil metod turli sinflarda o'ziga xos tarzda ishlaydi.

**Maqsad**: Kodni kengaytirish va yangilashni osonlashtirish. Polimorfizm yordamida interfeys yoki asosiy sinf orqali turli sinflar uchun bir xil metodlarni chaqirish mumkin bo'ladi.

**Farqi**: Polymorphism sinflar o'rtasida bir xil metod yoki interfeys orqali turli xatti-harakatlarni amalga oshirishga imkon beradi. Bunda metodlarning ishlash shakli turlicha bo'lsa ham, ularning chaqirilishi bir xil bo'ladi.

class Animal // Base class (parent)

{

public void animalSound()

{

Console.WriteLine("The animal makes a sound");

}

}

class Pig : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The pig says: wee wee");

}

}

class Dog : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The dog says: bow wow");

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

**4. Abstraction (Abstraksiya)**

**Tarif**: Abstraksiya orqali dasturchi faqat muhim bo'lgan ma'lumotlarni ajratib, ortiqcha tafsilotlarni yashiradi. Bu tamoyil abstrakt sinflar va interfeyslar yordamida amalga oshiriladi.

**Maqsad**: Murakkablikni kamaytirish va dasturiy ta'minotni soddalashtirish. Abstraksiya yordamida faqat asosiy funksiyalarni amalga oshirishga e'tibor qaratiladi va sinfning ichki ishlari yashiriladi.

**Farqi**: Abstraction dasturchiga faqat kerakli qismlarni ko'rsatib, qolgan tafsilotlarni yashirish imkonini beradi. Bu, ayniqsa, yirik dasturlar yaratishda murakkablikni kamaytirish uchun ishlatiladi.

abstract class Animal

{

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep()

{

Console.WriteLine("Zzz");

}

}

// Derived class (inherit from Animal)

class Pig : Animal

{

public override void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound(); // Call the abstract method

myPig.sleep(); // Call the regular method

}

}

**Xulosa**

|  |  |  |  |
| --- | --- | --- | --- |
| Tamoyil | Tarif | Maqsad | Farqi |
| Inheritance | Bir sinf boshqa sinf xususiyatlarini o'zlashtiradi | Kodni qayta ishlatish | Sinflar o'rtasida "bu shunday narsadir" munosabati |
| Encapsulation | Ma'lumotlarni himoya qilib, ularni boshqaradi | Ma'lumot xavfsizligi va noto'g'ri ishlatilishni oldini olish | Sinfdagi ma'lumotlarni boshqarishni ta'minlaydi |
| Polymorphism | Bir xil metod turli sinflarda turlicha ishlaydi | Kodni kengaytirishni osonlashtirish | Bir metod turli sinflarda o'ziga xos shaklda ishlaydi |
| Abstraction | Murakkablikni yashiradi va faqat kerakli qismlarni ko'rsatadi | Kodni soddalashtirish va o'zaro bog'liqlikni kamaytirish | Ortiqcha tafsilotlarni yashirishga yordam beradi |

Interface

**Interface** - bu sinf va strukturalarda turli xatti-harakatlar yoki qobiliyatlarni shablon tarzida belgilaydigan tuzilma. U bir yoki bir nechta sinflarga bir xil xatti-harakatlarni qo‘llash imkonini beradi va bu sinflar interface da ko‘rsatilgan metod va xususiyatlarni amalga oshirishi kerak.

**Maqsadi** - sinfning xatti-harakatlarini standartlashtirish, ko‘p tomonlama meros olishni ta'minlash, va polimorfizmni qo‘llab-quvvatlash.

**Abstract sinf** dan farqi - interface faqat metod va xususiyatlarning nomlarini beradi, lekin implementatsiya yo‘q.

public interface IHayvon

{

void OvozChiqazish();

}

public class It : IHayvon

{

public void OvozChiqazish() => Console.WriteLine("It hurmoqda");

}

public class Mushuk : IHayvon

{

public void OvozChiqazish() => Console.WriteLine("Mushuk miyovlamoqda");

}

var kalit so'zi o'zgaruvchilarni bilvosita e'lon qilish uchun ishlatiladi, bu kompilyatorga kompilyatsiya vaqtida o'zgaruvchining turini aniqlash imkonini beradi.

Foydalanish:

Misol:

var name = "Jon"; // satr sifatida qabul qilinadi

var yoshi = 25;

// int sifatida qabul qilinadi

var isActive = true; // bool deb xulosa qilinadi

Key Points:

● var faqat mahalliy miqyosda (usul, tsikl va boshqalar) ishlatilishi mumkin.

● Turni kompilyatsiya vaqtida aniqlash mumkin bo'lishi kerak.

● Turi aniq bo'lsa, o'qishni yaxshilaydi, lekin ortiqcha foydalanish kod ravshanligini kamaytirishi mumkin.

Enum

C# tilida **enum** (enumeration) — bu ma'lum bir qiymatlar to'plamini ifodalovchi maxsus tur. **enum** yordamida bir nechta doimiy qiymatlarni bir joyda saqlab, ularga oson kirish va ishlov berish mumkin. enum odatda, o'zgaruvchilarni o'zaro bog'lab, ular orasidagi aniq farqlarni belgilashda ishlatiladi.

Sintaksis:

enum EnumNomi

{

Qiymat1,

Qiymat2,

Qiymat3

}

enum Rang

{

Qizil,

Yashil,

Ko'k

}

Bu yerda, Rang enum turi 3 ta qiymatdan iborat: Qizil, Yashil, Ko'k. Har bir qiymat avtomatik ravishda raqamli qiymatga ega bo'ladi, boshlang'ich qiymat 0 dan boshlanadi va har bir keyingi qiymat 1 ga oshib boradi.

Misolni ishlatish:

Rang meningRangim = Rang.Qizil;

Console.WriteLine(meningRangim); // Qizil

Qisqacha xulosa:

**enum** turli doimiy qiymatlarni (odatda, bir xil kategoriyaga mansub bo'lgan) aniqlash uchun ishlatiladi.

Avtomatik tarzda raqamli qiymatlar oladi, lekin istalgan qiymatni berish mumkin.

**Null, ??, ??=, ?., ref, out, in, va params**

C# tilida quyidagi operatorlar va kalit so'zlar mavjud: **null, ??, ??=, ?., ref, out, in, va params**. Ularning har biri turli vazifalarni bajarishda ishlatiladi. Quyida ular haqida qisqacha tushuntirish:

1. null

null — bu qiymatsiz o'zgaruvchi yoki obyektni bildiradi. Agar o'zgaruvchi null bo'lsa, u hech qanday qiymatga ega emas.

string name = null; // name o'zgaruvchisi null qiymatiga ega

2. ?? **(Null-coalescing operator)**

?? operatori, agar birinchi operand null bo'lsa, ikkinchi operandni qaytaradi. Agar birinchi operand null bo'lmasa, u birinchi operandni qaytaradi.

string name = null;

string result = name ?? "No name"; // "No name" qaytariladi, chunki name null

3. ??= **(Null-coalescing assignment operator)**

??= operatori, agar o'zgaruvchi null bo'lsa, unga qiymat belgilaydi. Agar o'zgaruvchi null bo'lmasa, hech narsa o'zgarmaydi.

string name = null;

name ??= "Default name"; // name null bo'lgani uchun "Default name" bilan yangilanadi

4. ?. **(Null-conditional operator)**

?. operatori, agar obyekt null bo'lmasa, uning metodini yoki xususiyatini chaqiradi, aks holda hech narsa qilmaydi va null qaytaradi. Bu xususiyat ob'ektlar bilan ishlashda NullReferenceExceptionni oldini olishga yordam beradi.

string name = null;

int? length = name?.Length; // name null bo'lgani uchun length null bo'ladi

5. ref **(Reference keyword)**

ref kalit so'zi, metodga o'zgaruvchi qiymatini uzatishda ishlatiladi, lekin o'zgaruvchining qiymatini metod ichida o'zgartirish kerak bo'lganda. O'zgaruvchi metod chaqirilishidan oldin initsializatsiya qilinishi kerak.

public void UpdateValue(ref int value)

{

value = value + 1;

}

int num = 5;

UpdateValue(ref num); // num = 6

6. out **(Output keyword)**

out kalit so'zi metodga qiymatni uzatishda ishlatiladi, lekin o'zgaruvchining qiymati metod ichida belgilanishi kerak. out yordamida metoddan bir nechta qiymat qaytarish mumkin.

public void GetValues(out int a, out int b)

{

a = 10;

b = 20;

}

int x, y;

GetValues(out x, out y); // x = 10, y = 20

7. in **(Input keyword)**

in kalit so'zi o'zgaruvchi metodga faqat o'qish uchun uzatilishini ta'minlaydi, ya'ni metod ichida o'zgaruvchi qiymati o'zgarmaydi. Bu, asosan, katta hajmdagi ma'lumotlarni samarali tarzda uzatishda ishlatiladi.

public void DisplayValue(in int value)

{

Console.WriteLine(value);

}

int num = 10;

DisplayValue(in num); // num qiymati metodda o'zgarmaydi

8. params **(Parameter array)**

params kalit so'zi metodga bir nechta qiymatni massiv sifatida uzatishga imkon beradi. Bu yordamida bir nechta parametrlarni bitta massivga to'plash mumkin. Parametrlar massiv sifatida uzatilganda, ular metodga dinamik ravishda uzatiladi.

public void PrintNumbers(params int[] numbers)

{

foreach (var num in numbers)

{

Console.WriteLine(num);

}

}

PrintNumbers(1, 2, 3, 4, 5); // Barcha raqamlar ekranga chiqariladi

**Xulosa:**

|  |  |  |
| --- | --- | --- |
| Operator / Kalit so'z | Tushuncha | Misol |
| null | Qiymatsiz yoki obyektga nisbatan hech qanday qiymat. | string name = null; |
| ?? | Agar o'zgaruvchi null bo'lsa, boshqa qiymatni qaytaradi. | string name = null; string result = name ?? "No name"; |
| ??= | Agar o'zgaruvchi null bo'lsa, unga qiymat beriladi. | name ??= "Default"; |
| ?. | Agar ob'ekt null bo'lmasa, metod yoki xususiyatni chaqiradi. | var len = name?.Length; |
| ref | O'zgaruvchini metodga o'zgartirish uchun uzatadi. | UpdateValue(ref num); |
| out | Metoddan qaytadigan qiymatni uzatish uchun ishlatiladi. | GetValues(out a, out b); |
| in | O'zgaruvchini faqat o'qish uchun uzatadi. | DisplayValue(in num); |
| params | Bir nechta qiymatni massiv sifatida uzatadi. | PrintNumbers(1, 2, 3); |

Bu operatorlar va kalit so'zlar yordamida C# dasturlarida ma'lumotlar bilan samarali va xavfsiz ishlash imkoniyatlari yaratish mumkin.

**This** va **Static**

C# tilida **this** va **static** — bu ikki muhim kalit so'z bo'lib, ular ob'ektlar va sinflar bilan ishlashda alohida vazifalarni bajaradi. Quyida ular haqida qisqacha tushuntirish beraman.

1. this

this kalit so'zi, **hozirgi ob'ekt**ni anglatadi. this o'zgaruvchisi sinf yoki obyekt ichidagi metodlarda ishlatiladi va metod yoki konstruktorni chaqirgan obyektga murojaat qiladi. this yordamida o'zgaruvchilarga murojaat qilish yoki o'zgaruvchi nomlari bilan xatoliklarni oldini olish mumkin.

Asosiy ishlatilishi:

**Ob'ektning o'ziga murojaat qilish**: this yordamida ob'ektning xususiyatlarini yoki metodlarini chaqirish.

**Xususiyatlar va parametrlarga nisbatan nom to'qnashuvini hal qilish**.

Misol:

public class Mashina

{

public string model;

// Konstruktor

public Mashina(string model)

{

// Bu yerda 'model' - parametr, 'this.model' esa sinfning xususiyati

this.model = model;

}

public void ShowModel()

{

Console.WriteLine(this.model); // 'this' orqali sinfning xususiyatiga murojaat qilindi

}

}

var mashina = new Mashina("Tesla");

mashina.ShowModel(); // Tesla

Bu yerda this.model sinfning model xususiyatini anglatadi. model parametri bilan nom to'qnashuvini oldini olish uchun this ishlatilgan.

2. static

static kalit so'zi sinf darajasidagi (klass-level) xususiyatlar, metodlar yoki konstruktorlar uchun ishlatiladi. **static** xususiyatlar va metodlar ob'ektlar bilan bog'lanmagan bo'lib, ular sinfga tegishli. Ya'ni, static yordamida ob'ekt yaratmasdan sinfga to'g'ridan-to'g'ri murojaat qilish mumkin.

Asosiy ishlatilishi:

**Statik metodlar va xususiyatlar**: Ob'ekt yaratmasdan sinfning metodlarini yoki xususiyatlarini chaqirish uchun.

**Global ma'lumotlarga kirish**: Dasturda umumiy ma'lumotni saqlash va ulardan foydalanish.

Misol:

public class Calculator

{

// Statik metod

public static int Add(int a, int b)

{

return a + b;

}

}

int result = Calculator.Add(5, 3); // Ob'ekt yaratmasdan sinfning statik metodiga murojaat qildik

Console.WriteLine(result); // 8

Yuqoridagi misolda, Add metodiga to'g'ridan-to'g'ri sinf nomi orqali murojaat qilingan va ob'ekt yaratish shart emas.

Statik o'zgaruvchilar:

public class Counter

{

public static int count = 0;

public static void Increment()

{

count++;

}

}

Counter.Increment();

Console.WriteLine(Counter.count); // 1

Bu yerda count — statik o'zgaruvchi, uni sinf nomi orqali chaqirish mumkin. Statik metod va o'zgaruvchilar sinf darajasida mavjud bo'lib, ular sinfning har bir ob'ekti uchun umumiy bo'ladi.

Farqlar:

|  |  |  |
| --- | --- | --- |
| **Xususiyat** | this | static |
| **Qayerda ishlatiladi?** | Faqat instansiya metodlarida (obyekt metodlari) | Sinf darajasida, ob'ekt yaratmasdan ishlatiladi |
| **Nimani anglatadi?** | Hozirgi ob'ektni | Sinf darajasidagi metod yoki xususiyatni |
| **Ob'ekt yaratish kerakmi?** | Ha, this faqat ob'ekt mavjud bo'lganda ishlaydi | Yo'q, ob'ekt yaratmasdan ishlatiladi |
| **Misol** | this.model (hozirgi ob'ektning xususiyati) | Calculator.Add(5, 3) (statik metod chaqirildi) |

Xulosa:

**this**: Hozirgi ob'ektga murojaat qiladi, ob'ekt metodlarida ishlatiladi.

**static**: Ob'ektlar bilan bog'lanmagan sinf metodlarini va xususiyatlarini yaratadi, sinf nomi orqali chaqiriladi.

**Recursive funksiya**

**Recursive funksiya** — bu o'zini-o'zi chaqiradigan funksiya. Rekursiya yordamida biron bir masalani bir nechta kichik qismlarga bo'lib, ular orasida o'zini takrorlash orqali hal qilish mumkin. Rekursiv funksiya har safar o'zini chaqirganida masalani kichikroq va soddaroq holatda qayta ishlaydi, va oxir-oqibat masala eng oddiy holatga (baza holatiga) yetganda, funksiya to'xtaydi.

Rekursiya asosiy prinsipi:

**Baza holati (Base case)**: Rekursiya to'xtashi kerak bo'lgan holat. Agar baza holati mavjud bo'lmasa, rekursiya cheksiz ravishda davom etadi.

**Rekursiv chaqiruv (Recursive call)**: Funksiya o'zini o'zi chaqiradi, lekin bu chaqiruv har safar masalani kichikroq qilishga olib keladi.

Misol 1: Faktoriyel hisoblash

Faktoriyel (n!) — bu barcha natural sonlarning ko'paytmasi:

n! = n \* (n-1) \* (n-2) \* ... \* 1

Masalan, 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120.

Rekursiv funksiya bilan n! ni hisoblash quyidagicha bo'ladi:

public class Program

{

// Faktoriyelni rekursiv tarzda hisoblaydigan funksiya

public static int Factorial(int n)

{

// Baza holati: Agar n = 0 bo'lsa, 1 qaytariladi

if (n == 0)

return 1;

// Rekursiv chaqiruv: n \* (n-1)!

return n \* Factorial(n - 1);

}

public static void Main()

{

int result = Factorial(5);

Console.WriteLine(result); // 120

}

}

Izoh:

**Baza holati**: if (n == 0) return 1; — bu rekursiyaning to'xtash sharti, chunki 0! ning qiymati 1 ga teng.

**Rekursiv chaqiruv**: return n \* Factorial(n - 1); — bu rekursiv chaqiruv, unda Factorial funksiyasi o'zini qayta chaqiradi, lekin n har safar kichrayib boradi.

Misol 2: Fibonacci soni

Fibonacci sonlari — bu ketma-ketlik, unda har bir son oldingi ikkita sonning yig'indisiga teng bo'ladi:

F(0) = 0, F(1) = 1, va F(n) = F(n-1) + F(n-2) (n ≥ 2)

Rekursiv usulda Fibonacci sonini hisoblash:

public class Program

{

public static int Fibonacci(int n)

{

// Baza holati: F(0) va F(1)

if (n == 0)

return 0;

if (n == 1)

return 1;

// Rekursiv chaqiruv: F(n) = F(n-1) + F(n-2)

return Fibonacci(n - 1) + Fibonacci(n - 2);

}

public static void Main()

{

int result = Fibonacci(6); // F(6)

Console.WriteLine(result); // 8

}

}

Izoh:

**Baza holati**: if (n == 0) return 0; va if (n == 1) return 1; — bu Fibonacci ketma-ketligining boshlang'ich shartlari.

**Rekursiv chaqiruv**: return Fibonacci(n - 1) + Fibonacci(n - 2); — bu rekursiya orqali Fibonacci sonini hisoblash, har safar ikkita kichikroq Fibonacci sonlarini qo'shadi.

Rekursiv funktsiyaning afzalliklari:

Ba'zi masalalarni hal qilishda kodni soddalashtiradi.

Murakkab masalalarni kichikroq bo'laklarga bo'lish orqali hal qilish imkonini beradi (masalan, daraxt strukturalarida yoki qidiruv algoritmlarida).

Rekursiyaning kamchiliklari:

**Performance**: Rekursiya ko'pincha unchalik samarali emas, chunki har bir chaqiruvda yangi metodni chaqirish resurslarni talab qiladi (stack overflow xatoliklari).

**Baza holati muhim**: Baza holatini noto'g'ri belgilash rekursiyaning cheksiz davom etishiga olib kelishi mumkin.

Xulosa:

**Rekursiya** — bu o'zini o'zi chaqiradigan funksiya, masalani kichik qismlarga bo'lib hal qilish uchun ishlatiladi.

**Baza holati va rekursiv chaqiruv** — rekursiyaning muvaffaqiyatli ishlashi uchun zarur bo'lgan ikki asosiy elementdir.

Rekursiya ko'plab masalalarni hal qilishda qulay bo'lsa-da, uni samarali ishlatish uchun ehtiyotkorlik bilan baza holatini belgilash va qayta chaqiruvni optimallashtirish zarur.

**Boxing** va **Unboxing**

C# tilida **value types** (qiymat turlari) va **reference types** (ma'lumot turlari) o'rtasidagi farqlarni tushunishda muhim tushunchalardir. Bu jarayonlar C# dasturlash tilida ma'lumotlar turlarini o'zgartirish va ularga ishlov berish jarayonida qo'llaniladi.

1. **Boxing**

**Boxing** — bu qiymat turidagi o'zgaruvchini **ob'ekt turiga** aylantirish jarayonidir. Ya'ni, int, double, char kabi qiymat turlari **ob'ekt turiga** (masalan, object) o'zgartiriladi. Bu jarayon avtomatik ravishda amalga oshiriladi, lekin u uchun ortiqcha xotira sarfi va konvertatsiya (transformatsiya) talab qilinadi.

Misol:

int num = 123; // Value type (int)

object obj = num; // Boxing: int -> object

Bu yerda num qiymat turi (int) object turiga o'zgartirilmoqda. **Boxing** jarayoni, num qiymatining o'zini xotiradagi ob'ektga o'ramoqda (box qiladi).

2. **Unboxing**

**Unboxing** — bu **ob'ekt turidagi** ma'lumotni qaytadan **qiymat turiga** (masalan, int, double) o'zgartirish jarayonidir. Bu jarayon **explicit** (aniq) tarzda amalga oshiriladi va noto'g'ri unboxing xatoliklar (exception) keltirib chiqarishi mumkin, masalan, ob'ekt o'zgaruvchisi haqiqatan ham kerakli qiymat turiga ega bo'lmasa.

Misol:

object obj = 123; // Boxing: int -> object

int num = (int)obj; // Unboxing: object -> int

Bu yerda obj ob'ekt turi sifatida int qiymatini saqlaydi. obj dan int ga o'zgartirish (unboxing) qilinganida, uning qiymatini qayta ishlash mumkin.

Boxing va Unboxing o'rtasidagi farqlar:

|  |  |  |
| --- | --- | --- |
| **Xususiyat** | **Boxing** | **Unboxing** |
| **Jarayon** | Qiymat turidan ob'ekt turiga aylantirish. | Ob'ekt turidan qiymat turiga qaytarish. |
| **Avtomatikmi?** | Ha, C# tilida avtomatik tarzda amalga oshiriladi. | Yo'q, aniq tarzda (explicit) amalga oshirilishi kerak. |
| **Xotira sarfi** | Yangi ob'ekt yaratish, bu xotira sarfini oshiradi. | Hech qanday qo'shimcha xotira sarfi talab etilmaydi, lekin noto'g'ri unboxing xatoliklarni keltirib chiqarishi mumkin. |
| **Xatoliklar** | - | Agar noto'g'ri unboxing qilingan bo'lsa, InvalidCastException xatoligi yuzaga kelishi mumkin. |
| **Masala** | Qiymat turidagi ma'lumotni ob'ekt sifatida saqlash. | Ob'ektni asl qiymat turiga qaytarish. |

Boxing va Unboxingning samaradorligi

**Boxing** jarayonida qiymat turidagi o'zgaruvchi ob'ektga aylantirilganda, tizim yangi xotira maydoni ajratadi. Bu, ayniqsa ko'p sonli o'zgaruvchilarni box qilishda samaradorlikka ta'sir qilishi mumkin.

**Unboxing** jarayonida esa noto'g'ri turdagi ma'lumotga unboxing qilish xatoliklarni keltirib chiqarishi mumkin.

Xulosa:

**Boxing**: Qiymat turini ob'ekt turiga aylantirish. Bu jarayon avtomatik tarzda amalga oshiriladi, lekin xotira sarfi va ishlov berish uchun qo'shimcha resurslar talab qiladi.

**Unboxing**: Ob'ekt turini qiymat turiga qaytarish. Bu jarayon aniqlik bilan amalga oshirilishi kerak va noto'g'ri unboxing xatoliklarni keltirib chiqarishi mumkin.

**Generic** va **Delegate**

**Generic** va **Delegate** — C# tilida yuqori darajadagi kontseptsiyalar bo'lib, ular kodni yanada moslashuvchan va qayta foydalanish imkonini beradi. Ular yordamida turli turlar bilan ishlashda samaradorlikni oshirish, yoki funksiyalarga parametr sifatida metodlarni uzatish mumkin.

1. **Generic**

**Generic** — bu turli xil turdagi (types) ma'lumotlar bilan ishlash imkonini beradigan mexanizmdir. **Generic** yordamida siz bir xil algoritmlarni yoki metodlarni turli turdagi ma'lumotlar bilan ishlatish imkoniyatiga egasiz. Bu kodni takrorlashdan saqlaydi va turli ma'lumot turlariga moslashuvchanlik beradi.

Asosiy tushuncha:

**Generic** yordamida, siz metodlar yoki sinflar yaratishda turlarni aniqlamaysiz, balki ularga parametr sifatida turni uzatishingiz mumkin.

**Type safety**: Generic, turlarning mosligini tekshiradi, bu esa runtime xatoliklaridan saqlanishga yordam beradi.

Misol: Generic metod

public class Printer

{

// Generic metod

public void Print<T>(T item)

{

Console.WriteLine(item);

}

}

public class Program

{

public static void Main()

{

Printer printer = new Printer();

// Int turi bilan ishlash

printer.Print(123); // 123

// String turi bilan ishlash

printer.Print("Hello World!"); // Hello World!

}

}

Bu yerda Print metodi generik bo'lib, T parametri yordamida istalgan turdagi ma'lumotni qabul qilish imkonini beradi.

Misol: Generic sinf

public class Box<T>

{

private T \_value;

public void SetValue(T value)

{

\_value = value;

}

public T GetValue()

{

return \_value;

}

}

public class Program

{

public static void Main()

{

Box<int> intBox = new Box<int>();

intBox.SetValue(10);

Console.WriteLine(intBox.GetValue()); // 10

Box<string> stringBox = new Box<string>();

stringBox.SetValue("Generics");

Console.WriteLine(stringBox.GetValue()); // Generics

}

}

Genericning afzalliklari:

**Type safety**: Turli turlarning to'g'riligini kompilyatsiya vaqtida tekshiradi.

**Kodni qayta ishlatish**: Bir nechta turdagi ma'lumotlar bilan ishlash imkonini beradi, kodni takrorlashdan saqlaydi.

**Samarlilik**: Generic metodlar va sinflar turli turlar bilan ishlashda yuqori samaradorlikni ta'minlaydi.

2. **Delegate**

**Delegate** — bu metodni ko'rsatadigan tur (type-safe function pointer) bo'lib, boshqa metodlarni o'zgaruvchilar sifatida saqlash va ularni bajarishga imkon beradi. Delegate yordamida siz biror metodni parametr sifatida uzatishingiz yoki metodni keyinroq chaqirishingiz mumkin.

Asosiy tushuncha:

**Delegate** yordamida metodni uzatish va bajarish mumkin, bu esa hodisalar (events) va metodlarni dinamik ravishda chaqirishda foydalidir.

Delegate ob'ektga o'xshash bo'lib, metodga bog'langan.

Misol: Delegate deklaratsiyasi va ishlatish

public delegate void MyDelegate(string message);

public class Program

{

// Delegate orqali metodni uzatish

public static void DisplayMessage(string message)

{

Console.WriteLine(message);

}

public static void Main()

{

// Delegate ob'ektini yaratish va metodga bog'lash

MyDelegate del = new MyDelegate(DisplayMessage);

// Delegate orqali metodni chaqirish

del("Hello from Delegate!"); // Hello from Delegate!

}

}

Bu yerda MyDelegate — metodni ko'rsatadigan delegate turidir. DisplayMessage metodi bu delegate orqali chaqiriladi.

Misol: Multiple Delegate va Chaqirish

Delegatelar bir nechta metodlarga bog'lanishi mumkin. Bu usul **multicast delegate** deb ataladi.

public delegate void MyDelegate(string message);

public class Program

{

public static void DisplayMessage(string message)

{

Console.WriteLine(message);

}

public static void DisplayUpperMessage(string message)

{

Console.WriteLine(message.ToUpper());

}

public static void Main()

{

MyDelegate del = DisplayMessage;

del += DisplayUpperMessage; // Ikki metodni bog'lash

del("Hello from Delegate!"); // Output: Hello from Delegate! / HELLO FROM DELEGATE!

}

}

Bu yerda del delegate orqali ikkita metod ( DisplayMessage va DisplayUpperMessage ) chaqiriladi.

Delegatening afzalliklari:

**Flexibility**: Delegate yordamida metodlarni parametr sifatida uzatish va keyinroq chaqirish mumkin.

**Event handling**: Delegate hodisalar bilan ishlashda keng qo'llaniladi, chunki hodisalar metodlarni dinamik ravishda chaqirishi mumkin.

**Multiple methods**: Bir nechta metodni bitta delegatega bog'lash va ularni ketma-ket chaqirish imkoniyati mavjud.

Xulosa:

**Generic**: Bu turli turdagi ma'lumotlar bilan ishlash imkonini beruvchi mexanizm. Generic yordamida bir xil kodni turli turlar uchun ishlatish mumkin, bu esa kodni qayta ishlatishga yordam beradi.

**Delegate**: Bu metodlarni parametr sifatida uzatish va chaqirish imkonini beruvchi mexanizm. Delegate hodisalar va metodlarni dinamik ravishda chaqirishda ishlatiladi.

As va Is

C# tilida **as** va **is** operatorlari turlarni tekshirish va konvertatsiya qilishda ishlatiladi. Har ikkalasi ham tur tekshiruvlari va ma'lumotlarni xavfsiz tarzda o'zgartirishda qo'llaniladi, lekin ular turli maqsadlarda ishlatiladi.

1. is **operatori**

**is** operatori — bu ob'ektning ma'lum bir turga mos kelishini tekshiradigan operator. U ob'ektning turini tekshirib, to'g'ri yoki noto'g'ri (true yoki false) natija beradi. is operatori, agar tur mos kelsa, true qaytaradi, aks holda false qaytaradi.

Sintaksis:

object obj = ...;

if (obj is SomeType) {

// SomeType ga mos keladi

}

Misol:

public class Animal { }

public class Dog : Animal { }

public class Program

{

public static void Main()

{

Animal animal = new Dog();

if (animal is Dog) // Turini tekshirish

{

Console.WriteLine("This is a Dog!");

}

else

{

Console.WriteLine("This is not a Dog.");

}

}

}

Bu yerda animal is Dog ifodasi true qaytaradi, chunki animal ob'ekti aslida Dog turiga ega.

2. as **operatori**

**as** operatori — bu tur konvertatsiyasini xavfsiz tarzda amalga oshiradi. Agar ma'lumot turini muvaffaqiyatli tarzda o'zgartira olmasa, null qaytaradi. Bu operator **null** qiymatini qaytarish orqali konvertatsiya xatoliklaridan saqlanishga yordam beradi.

Sintaksis:

SomeType result = obj as SomeType;

if (result != null)

{

// result ob'ekti muvaffaqiyatli turga o'zgartirildi

}

else

{

// obj ob'ekti SomeType turiga o'zgartirilolmadi

}

Misol:

public class Animal { }

public class Dog : Animal { }

public class Program

{

public static void Main()

{

Animal animal = new Dog();

// as operatoridan foydalanish

Dog dog = animal as Dog;

if (dog != null) // Agar muvaffaqiyatli o'zgartirildi

{

Console.WriteLine("This is a Dog!");

}

else

{

Console.WriteLine("This is not a Dog.");

}

}

}

Bu yerda animal as Dog ifodasi animal ob'ektini Dog turiga o'zgartirishga harakat qiladi. Agar animal ob'ekti Dog turiga mos bo'lsa, u holda dog o'zgaruvchisi Dog turidagi ob'ektni oladi. Agar animal ob'ekti Dog turiga mos kelmasa, dog null bo'ladi.

is va as operatorlarining farqlari:

|  |  |  |
| --- | --- | --- |
| **Xususiyat** | is | as |
| **Maqsadi** | Tur mosligini tekshirish. | Turga xavfsiz konvertatsiya qilish. |
| **Natija** | true yoki false | Konvertatsiya qilingan ob'ekt yoki null |
| **Xatoliklar** | Agar tur mos kelmasa, false qaytaradi. | Agar konvertatsiya amalga oshmasa, null qaytaradi. |
| **Tavsiya etilgan foydalanish** | Turni tekshirish va shartlar yaratishda. | Turni xavfsiz tarzda o'zgartirish. |

Xulosa:

**is** operatori ob'ektning turini tekshirish uchun ishlatiladi va bu to'g'ri yoki noto'g'ri natija qaytaradi.

**as** operatori esa ob'ektni belgilangan turga xavfsiz tarzda o'zgartiradi va agar bu mumkin bo'lmasa, null qaytaradi.

Shuning uchun, agar siz ob'ektning turini faqat tekshirmoqchi bo'lsangiz, is operatoridan, agar turini xavfsiz tarzda o'zgartirmoqchi bo'lsangiz, as operatoridan foydalaning.

ASP.NET Core Web API

\*\*ASP.NET Core Web API\*\* — bu HTTP protokoli orqali turli mijozlar (client) bilan muloqot qiladigan va ma'lumotlarni yuborish, olish, yangilash yoki o'chirishni ta'minlaydigan web-servislarni yaratishga mo'ljallangan framework. ASP.NET Core — bu Microsoft tomonidan ishlab chiqilgan ochiq manba kodli platforma bo'lib, asosan yuqori samaradorlikka ega, multiplatform (Windows, Linux, macOS) bo'lib ishlaydi.

Web API (Application Programming Interface) — bu boshqa dasturlar bilan o'zaro aloqada bo'lish imkonini beruvchi interfeys. ASP.NET Core Web API yordamida turli ma'lumotlar bazalari, tashqi tizimlar yoki front-end ilovalar bilan muloqot qilish mumkin.

### \*\*ASP.NET Core Web API ning asosiy tushunchalari\*\*

1. \*\*API (Application Programming Interface)\*\*:

API — bu dastur komponentlari o'rtasida o'zaro aloqalarni amalga oshiradigan interfeys. Web API esa aynan HTTP protokoli orqali ishlaydi va asosan JSON yoki XML formatidagi ma'lumotlarni uzatadi.

2. \*\*RESTful API\*\*:

REST (Representational State Transfer) — bu HTTP metodlari (GET, POST, PUT, DELETE) orqali resurslar bilan ishlashni ta'minlovchi arxitektura printsipi. ASP.NET Core Web API odatda \*\*RESTful\*\* xizmatlarni yaratish uchun ishlatiladi.

3. \*\*Controller\*\*:

Web API da mijozdan kelgan so'rovlarga javob beradigan logika \*\*Controller\*\*larda joylashadi. Har bir controller bir nechta \*\*action\*\* metodlaridan iborat bo'ladi. Bu metodlar HTTP so'rovlarini qabul qiladi va ularga javob beradi.

4. \*\*Routes\*\*:

Web API URL-lari \*\*routing\*\* yordamida aniqlanadi. Har bir controller va uning metodlari URL orqali chaqiriladi. Misol uchun, `GET /api/products` orqali `ProductsController` ning `Get` metodiga murojaat qilinadi.

5. \*\*Action Methods\*\*:

Controllerda joylashgan metodlar \*\*action methods\*\* deb ataladi. Har bir action HTTP metodlariga (GET, POST, PUT, DELETE) mos keladi. Action metodlar ma'lumotni qaytarish, yangilash yoki o'chirishni amalga oshiradi.

6. \*\*Model\*\*:

Model — bu Web API orqali yuboriladigan yoki qabul qilinadigan ma'lumotlarni ifodalovchi klasslardir. Model orqali ma'lumotlar strukturasi (masalan, foydalanuvchi, mahsulot) belgilanadi.

7. \*\*Middleware\*\*:

Middleware — bu ASP.NET Core ilovasining bir qismini tashkil etuvchi va HTTP so'rovini qayta ishlovchi komponentlardir. Masalan, autentifikatsiya, logging, va error handling uchun middleware ishlatiladi.

### \*\*ASP.NET Core Web API arxitekturasi\*\*

1. \*\*HTTP Request\*\*: Mijozdan HTTP so'rovi keladi (masalan, GET, POST).

2. \*\*Routing\*\*: So'rov URL-ga qarab controller va metodga yo'naltiriladi.

3. \*\*Controller\*\*: Controller so'rovni qabul qiladi va kerakli logikani bajaradi.

4. \*\*Action Method\*\*: Action metodlar so'rovni qayta ishlaydi (ma'lumot olish, yangilash, o'chirish).

5. \*\*Model\*\*: Action metodlar model (ma'lumotlar) bilan ishlaydi.

6. \*\*HTTP Response\*\*: Javob HTTP orqali mijozga qaytariladi (odatda JSON formatida).

### \*\*ASP.NET Core Web API Misol\*\*

#### 1. \*\*Proyekt yaratish\*\*

1. \*\*Visual Studio\*\* orqali yangi ASP.NET Core Web API proyektini yaratish.

- "File" > "New" > "Project" > "ASP.NET Core Web Application".

- "API" shablonini tanlash.

2. \*\*Program.cs\*\* va \*\*Startup.cs\*\* fayllari avtomatik tarzda yaratiladi (agar eski versiyalarda ishlayotgan bo'lsangiz). Yangi versiyalar uchun faqat `Program.cs` fayli kerak.

#### 2. \*\*Controller yaratish\*\*

Controller — bu Web API ning markaziy qismidir. Har bir controller HTTP so'rovlarini qabul qiladi va ularga javob beradi.

\*\*ProductsController\*\* misoli:

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

namespace WebApiExample.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

// Simulyatsiya uchun list

private static List<string> products = new List<string> { "Product 1", "Product 2", "Product 3" };

// GET: api/products

[HttpGet]

public IActionResult GetProducts()

{

return Ok(products); // JSON formatida ro'yxatni qaytaradi

}

// GET: api/products/1

[HttpGet("{id}")]

public IActionResult GetProduct(int id)

{

if (id < 0 || id >= products.Count)

return NotFound(); // Agar id mavjud bo'lmasa, 404 qaytaradi

return Ok(products[id]); // JSON formatida bitta mahsulotni qaytaradi

}

// POST: api/products

[HttpPost]

public IActionResult CreateProduct([FromBody] string product)

{

products.Add(product); // Yangi mahsulot qo'shish

return CreatedAtAction(nameof(GetProduct), new { id = products.Count - 1 }, product); // Yangi yaratgan mahsulotni qaytaradi

}

}

}

```

- `HttpGet`, `HttpPost`, `HttpPut`, va `HttpDelete` kabi atributlar yordamida HTTP metodlari aniqlanadi.

- \*\*[Route]\*\* atributi orqali controllerning URL-ni belgilash mumkin.

- \*\*Ok()\*\*, \*\*NotFound()\*\*, \*\*CreatedAtAction()\*\* kabi metodlar HTTP javoblarini qaytaradi.

3. \*\*Model yaratish\*\*

Model — bu API orqali uzatiladigan yoki qabul qilinadigan ma'lumotlarni tashkil etadi.

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

}

4. \*\*Routing\*\*

Routing ASP.NET Core Web API ning URL-manzillarini belgilash uchun ishlatiladi. Yuqoridagi misolda, `/api/products` va `/api/products/{id}` URL-lari `ProductsController`ga bog'langan.

### \*\*Ma'lumotlar Bazasiga ulanish\*\*

Web API ko'pincha ma'lumotlar bazasiga ulanishni talab qiladi. Bu uchun Entity Framework yoki boshqa ORM (Object Relational Mapping) texnologiyalaridan foydalanish mumkin.

### \*\*Test qilish\*\*

1. \*\*Postman\*\* yoki \*\*Swagger\*\* kabi vositalar yordamida API ni test qilish mumkin.

2. \*\*Swagger\*\* API uchun avtomatik dokumentatsiya yaratadi va uni interaktiv tarzda test qilish imkonini beradi. ASP.NET Core da Swagger konfiguratsiyasini `Startup.cs` yoki `Program.cs` faylida qilish mumkin.

```csharp

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.ConfigureServices(services =>

{

services.AddControllers();

services.AddEndpointsApiExplorer();

services.AddSwaggerGen(); // Swagger qo'shish

})

.Configure(app =>

{

app.UseSwagger(); // Swagger ni ishga tushirish

app.UseSwaggerUI(); // Swagger UI

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

});

});

}

```

### \*\*Xulosa\*\*

- \*\*ASP.NET Core Web API\*\* yuqori samaradorlikka ega, multiplatforma qo'llab-quvvatlaydigan, va RESTful xizmatlarni yaratish uchun qulay frameworkdir.

- \*\*Controller\*\*lar yordamida HTTP so'rovlariga javob qaytariladi, \*\*Model\*\*lar yordamida ma'lumotlar struktura qilinadi, va \*\*Routing\*\* yordamida URL'lar belgilangan bo'ladi.

\*\*ASP.NET Core Web API\*\* — bu HTTP protokoli orqali turli mijozlar (client) bilan muloqot qiladigan va ma'lumotlarni yuborish, olish, yangilash yoki o'chirishni ta'minlaydigan web-servislarni yaratishga mo'ljallangan framework. ASP.NET Core — bu Microsoft tomonidan ishlab chiqilgan ochiq manba kodli platforma bo'lib, asosan yuqori samaradorlikka ega, multiplatform (Windows, Linux, macOS) bo'lib ishlaydi.

Web API (Application Programming Interface) — bu boshqa dasturlar bilan o'zaro aloqada bo'lish imkonini beruvchi interfeys. ASP.NET Core Web API yordamida turli ma'lumotlar bazalari, tashqi tizimlar yoki front-end ilovalar bilan muloqot qilish mumkin.

### \*\*ASP.NET Core Web API ning asosiy tushunchalari\*\*

1. \*\*API (Application Programming Interface)\*\*:

API — bu dastur komponentlari o'rtasida o'zaro aloqalarni amalga oshiradigan interfeys. Web API esa aynan HTTP protokoli orqali ishlaydi va asosan JSON yoki XML formatidagi ma'lumotlarni uzatadi.

2. \*\*RESTful API\*\*:

REST (Representational State Transfer) — bu HTTP metodlari (GET, POST, PUT, DELETE) orqali resurslar bilan ishlashni ta'minlovchi arxitektura printsipi. ASP.NET Core Web API odatda \*\*RESTful\*\* xizmatlarni yaratish uchun ishlatiladi.

3. \*\*Controller\*\*:

Web API da mijozdan kelgan so'rovlarga javob beradigan logika \*\*Controller\*\*larda joylashadi. Har bir controller bir nechta \*\*action\*\* metodlaridan iborat bo'ladi. Bu metodlar HTTP so'rovlarini qabul qiladi va ularga javob beradi.

4. \*\*Routes\*\*:

Web API URL-lari \*\*routing\*\* yordamida aniqlanadi. Har bir controller va uning metodlari URL orqali chaqiriladi. Misol uchun, `GET /api/products` orqali `ProductsController` ning `Get` metodiga murojaat qilinadi.

5. \*\*Action Methods\*\*:

Controllerda joylashgan metodlar \*\*action methods\*\* deb ataladi. Har bir action HTTP metodlariga (GET, POST, PUT, DELETE) mos keladi. Action metodlar ma'lumotni qaytarish, yangilash yoki o'chirishni amalga oshiradi.

6. \*\*Model\*\*:

Model — bu Web API orqali yuboriladigan yoki qabul qilinadigan ma'lumotlarni ifodalovchi klasslardir. Model orqali ma'lumotlar strukturasi (masalan, foydalanuvchi, mahsulot) belgilanadi.

7. \*\*Middleware\*\*:

Middleware — bu ASP.NET Core ilovasining bir qismini tashkil etuvchi va HTTP so'rovini qayta ishlovchi komponentlardir. Masalan, autentifikatsiya, logging, va error handling uchun middleware ishlatiladi.

### \*\*ASP.NET Core Web API arxitekturasi\*\*

1. \*\*HTTP Request\*\*: Mijozdan HTTP so'rovi keladi (masalan, GET, POST).

2. \*\*Routing\*\*: So'rov URL-ga qarab controller va metodga yo'naltiriladi.

3. \*\*Controller\*\*: Controller so'rovni qabul qiladi va kerakli logikani bajaradi.

4. \*\*Action Method\*\*: Action metodlar so'rovni qayta ishlaydi (ma'lumot olish, yangilash, o'chirish).

5. \*\*Model\*\*: Action metodlar model (ma'lumotlar) bilan ishlaydi.

6. \*\*HTTP Response\*\*: Javob HTTP orqali mijozga qaytariladi (odatda JSON formatida).

### \*\*ASP.NET Core Web API Misol\*\*

#### 1. \*\*Proyekt yaratish\*\*

1. \*\*Visual Studio\*\* orqali yangi ASP.NET Core Web API proyektini yaratish.

- "File" > "New" > "Project" > "ASP.NET Core Web Application".

- "API" shablonini tanlash.

2. \*\*Program.cs\*\* va \*\*Startup.cs\*\* fayllari avtomatik tarzda yaratiladi (agar eski versiyalarda ishlayotgan bo'lsangiz). Yangi versiyalar uchun faqat `Program.cs` fayli kerak.

#### 2. \*\*Controller yaratish\*\*

Controller — bu Web API ning markaziy qismidir. Har bir controller HTTP so'rovlarini qabul qiladi va ularga javob beradi.

\*\*ProductsController\*\* misoli:

```csharp

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

namespace WebApiExample.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

// Simulyatsiya uchun list

private static List<string> products = new List<string> { "Product 1", "Product 2", "Product 3" };

// GET: api/products

[HttpGet]

public IActionResult GetProducts()

{

return Ok(products); // JSON formatida ro'yxatni qaytaradi

}

// GET: api/products/1

[HttpGet("{id}")]

public IActionResult GetProduct(int id)

{

if (id < 0 || id >= products.Count)

return NotFound(); // Agar id mavjud bo'lmasa, 404 qaytaradi

return Ok(products[id]); // JSON formatida bitta mahsulotni qaytaradi

}

// POST: api/products

[HttpPost]

public IActionResult CreateProduct([FromBody] string product)

{

products.Add(product); // Yangi mahsulot qo'shish

return CreatedAtAction(nameof(GetProduct), new { id = products.Count - 1 }, product); // Yangi yaratgan mahsulotni qaytaradi

}

}

}

```

- `HttpGet`, `HttpPost`, `HttpPut`, va `HttpDelete` kabi atributlar yordamida HTTP metodlari aniqlanadi.

- \*\*[Route]\*\* atributi orqali controllerning URL-ni belgilash mumkin.

- \*\*Ok()\*\*, \*\*NotFound()\*\*, \*\*CreatedAtAction()\*\* kabi metodlar HTTP javoblarini qaytaradi.

#### 3. \*\*Model yaratish\*\*

Model — bu API orqali uzatiladigan yoki qabul qilinadigan ma'lumotlarni tashkil etadi.

```csharp

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

}

```

#### 4. \*\*Routing\*\*

Routing ASP.NET Core Web API ning URL-manzillarini belgilash uchun ishlatiladi. Yuqoridagi misolda, `/api/products` va `/api/products/{id}` URL-lari `ProductsController`ga bog'langan.

### \*\*Ma'lumotlar Bazasiga ulanish\*\*

Web API ko'pincha ma'lumotlar bazasiga ulanishni talab qiladi. Bu uchun Entity Framework yoki boshqa ORM (Object Relational Mapping) texnologiyalaridan foydalanish mumkin.

### \*\*Test qilish\*\*

1. \*\*Postman\*\* yoki \*\*Swagger\*\* kabi vositalar yordamida API ni test qilish mumkin.

2. \*\*Swagger\*\* API uchun avtomatik dokumentatsiya yaratadi va uni interaktiv tarzda test qilish imkonini beradi. ASP.NET Core da Swagger konfiguratsiyasini `Startup.cs` yoki `Program.cs` faylida qilish mumkin.

```csharp

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.ConfigureServices(services =>

{

services.AddControllers();

services.AddEndpointsApiExplorer();

services.AddSwaggerGen(); // Swagger qo'shish

})

.Configure(app =>

{

app.UseSwagger(); // Swagger ni ishga tushirish

app.UseSwaggerUI(); // Swagger UI

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

});

});

}

```

### \*\*Xulosa\*\*

- \*\*ASP.NET Core Web API\*\* yuqori samaradorlikka ega, multiplatforma qo'llab-quvvatlaydigan, va RESTful xizmatlarni yaratish uchun qulay frameworkdir.

- \*\*Controller\*\*lar yordamida HTTP so'rovlariga javob qaytariladi, \*\*Model\*\*lar yordamida ma'lumotlar struktura qilinadi, va \*\*Routing\*\* yordamida URL'lar belgilangan bo'ladi.

\*\*ASP.NET Core Web API\*\* yaratishda ishlatiladigan asosiy tushunchalar quyidagilar:

### 1. \*\*Controller\*\*

- \*\*Controller\*\* — bu Web API ning markaziy qismi bo'lib, HTTP so'rovlarini qabul qiladi va ularga javob beradi. Har bir controller bir nechta action metodlaridan iborat bo'ladi.

- \*\*Route\*\* orqali HTTP so'rovi qaysi controllerga yo'naltirilishini belgilash mumkin.

\*\*Misol\*\*: `ProductsController`, `UsersController`.

### 2. \*\*Action Method\*\*

- \*\*Action Method\*\* — bu controller ichidagi metodlar bo'lib, ular HTTP so'rovlariga (GET, POST, PUT, DELETE) javob beradi. Har bir action method ma'lum bir resurs yoki ma'lumotlar bilan ishlaydi.

- Masalan, `GetProduct()`, `CreateProduct()` metodlari.

### 3. \*\*Routing\*\*

- \*\*Routing\*\* — bu so'rovlar URL orqali controller va metodlarga yo'naltirilishini boshqaradi. URL va controller o'rtasida bog'lanish o'rnatish uchun \*\*[Route]\*\* atributidan foydalaniladi.

- Masalan, `api/products` — bu URL `ProductsController` ga tegishli.

### 4. \*\*Model\*\*

- \*\*Model\*\* — bu ma'lumotlar tuzilmasini ifodalovchi class bo'lib, Web API orqali yuboriladigan yoki qabul qilinadigan ma'lumotlarni tashkil etadi. Model odatda ma'lumotlar bazasi yoki foydalanuvchi interfeysi bilan bog'liq bo'ladi.

- Masalan, `Product`, `User` model klasslari.

### 5. \*\*HTTP Verbs (GET, POST, PUT, DELETE)\*\*

- Web API da ishlatiladigan asosiy HTTP metodlari:

- \*\*GET\*\* — ma'lumot olish.

- \*\*POST\*\* — yangi ma'lumot yaratish.

- \*\*PUT\*\* — mavjud ma'lumotni yangilash.

- \*\*DELETE\*\* — ma'lumotni o'chirish.

### 6. \*\*HTTP Response\*\*

- \*\*HTTP Response\*\* — bu serverdan mijozga yuboriladigan javob. Response ma'lumotlar (odatda JSON formatida) va HTTP status kodi (masalan, `200 OK`, `404 Not Found`)ni o'z ichiga oladi.

- Masalan, `Ok()`, `NotFound()`, `CreatedAtAction()` kabi metodlar orqali javob qaytariladi.

### 7. \*\*Middleware\*\*

- \*\*Middleware\*\* — bu HTTP so'rovlarini qayta ishlovchi komponentlar bo'lib, ular HTTP pipeline ichida ishlaydi. Middleware ko'pincha autentifikatsiya, xatoliklarni qayta ishlash, logging kabi vazifalarni bajaradi.

- Masalan, autentifikatsiya yoki caching middleware.

### 8. \*\*Swagger\*\*

- \*\*Swagger\*\* — bu API ning avtomatik dokumentatsiyasini yaratish uchun ishlatiladi. Swagger UI yordamida Web API ni interaktiv tarzda test qilish mumkin.

- ASP.NET Core da Swagger ni `AddSwaggerGen()` va `UseSwaggerUI()` yordamida o'rnatish mumkin.

### 9. \*\*Dependency Injection\*\*

- \*\*Dependency Injection\*\* (DI) — bu ASP.NET Core ning asosiy printsipi bo'lib, servislarni (masalan, database, autentifikatsiya, logging) controller yoki boshqa komponentlarga avtomatik tarzda injektsiya qilish imkonini beradi.

- DI yordamida xizmatlarni `ConfigureServices` metodida ro'yxatdan o'tkazish mumkin.

### 10. \*\*Status Codes\*\*

- \*\*HTTP Status Codes\*\* — bu server tomonidan javob berilganda qaytariladigan raqamlar, masalan:

- `200 OK` — so'rov muvaffaqiyatli bajarildi.

- `201 Created` — yangi resurs yaratildi.

- `400 Bad Request` — noto'g'ri so'rov.

- `404 Not Found` — resurs topilmadi.

- `500 Internal Server Error` — serverda xatolik yuz berdi.

### 11. \*\*Model Binding and Validation\*\*

- \*\*Model Binding\*\* — bu ASP.NET Core Web API da foydalanuvchidan yuborilgan ma'lumotlarni (JSON, XML) modelga avtomatik tarzda o'zgartirish jarayonidir.

- \*\*Model Validation\*\* — ma'lumotlar kiritilgan modelni tekshirish va unga to'g'ri qiymatlar kiritilganligini tasdiqlash.

### 12. \*\*Asynchronous Programming (Async/Await)\*\*

- \*\*Async/Await\*\* — Web API ning samaradorligini oshirish uchun asinxron metodlardan foydalanish imkonini beradi. Bu, ayniqsa, ma'lumotlar bazasiga so'rov yuborishda va tarmoq orqali ma'lumot yuborishda foydalidir.

### 13. \*\*Authentication and Authorization\*\*

- \*\*Authentication\*\* — bu foydalanuvchining kimligini aniqlash jarayoni.

- \*\*Authorization\*\* — bu foydalanuvchining ma'lum bir resurslarga kirish huquqini tekshirish jarayoni.

- ASP.NET Core Web API da bu uchun \*\*JWT (JSON Web Token)\*\* yoki \*\*OAuth\*\* kabi mexanizmlar ishlatiladi.

### Xulosa:

ASP.NET Core Web API asosida qurilgan tizimlar HTTP so'rovlari va javoblari orqali resurslarni boshqarishga imkon beradi. Controllerlar, routing, action metodlari va model binding kabi tushunchalar bu tizimlarning samarali ishlashini ta'minlaydi. Swagger yordamida esa API'ni avtomatik hujjatlashtirish va test qilish mumkin.

Asinxron dasturlash

\*\*Asinxron dasturlash\*\* — bu dasturda bir vaqtning o'zida bir nechta ishlarni bajarishga imkon beruvchi yondashuvdir. Bu yondashuv, dastur bir ishni bajarish bilan boshqa ishni kutish o'rniga, ishni bajarishni davom ettirib, kutayotgan paytda boshqa ishlarni amalga oshiradi. Asinxron dasturlash, ayniqsa, tarmoq so'rovlari, ma'lumotlar bazasi bilan ishlash va fayllarni o'qish/kiritish kabi uzoq davom etadigan vazifalarda samarali bo'ladi.

### \*\*Asinxron dasturlash nima?\*\*

Asinxron dasturlashda, dasturga berilgan buyruqlar ketma-ket bajarilishi o'rniga, bir vaqtning o'zida ko'plab vazifalar bajarilishi mumkin. Buning uchun \*\*asynchronous (async)\*\* va \*\*await\*\* kalit so'zlari ishlatiladi.

### \*\*Asinxron dasturlashning asosiy tushunchalari:\*\*

1. \*\*Asinxron metod (Async Method)\*\*:

- \*\*Async\*\* kalit so'zi yordamida metod asinxron sifatida belgilanishi mumkin. Asinxron metodlar vaqt oluvchi operatsiyalarni bajarishda yordam beradi (masalan, ma'lumotlar bazasiga so'rov yuborish, faylni o'qish).

- \*\*Async\*\* metodlar \*\*Task\*\* yoki \*\*Task<T>\*\* tipini qaytaradi.

\*\*Misol\*\*:

public async Task<int> GetDataAsync()

{

// Asinxron vazifani bajarish

await Task.Delay(2000); // 2 sekund kutish

return 42;

}

```

2. \*\*Await\*\*:

- \*\*Await\*\* kalit so'zi asinxron metodda ishlatiladi va metodning bajarilishini kutishga imkon beradi, lekin boshqa vazifalar bajarilishiga to'sqinlik qilmaydi.

- \*\*Await\*\* yordamida kodni sinxron ko'rinishda yozish mumkin, lekin bajarilishi asinxron bo'ladi.

\*\*Misol\*\*:

```csharp

public async Task ProcessDataAsync()

{

int result = await GetDataAsync(); // GetDataAsync metodini kutadi

Console.WriteLine(result);

}

```

3. \*\*Task\*\*:

- \*\*Task\*\* — bu asinxron metodlarning bajarilishini kutadigan obyekt. Task metodning bajarilishini ko'rsatadi va natijani qaytarish uchun ishlatiladi.

- \*\*Task<T>\*\* — bu asinxron metoddan qaytariladigan qiymatni ko'rsatadi. T — metodning qaytargan qiymati turi.

\*\*Misol\*\*:

```csharp

public async Task<string> GetDataAsync()

{

// Simulyatsiya uchun ma'lumot olish

await Task.Delay(1000); // 1 sekund kutish

return "Data received";

}

```

4. \*\*Asinxron metodning foydalari\*\*:

- \*\*Resurslar samaradorligini oshiradi\*\*: Asinxron metodlar CPU resurslarini samarali ishlatadi, chunki ular kutish jarayonida boshqa ishlarni bajarishga imkon beradi.

- \*\*Tezlikni oshiradi\*\*: Asinxron metodlar vaqtni qisqartiradi, chunki ular kutish vaqtida boshqa ishlarni bajaradi.

- \*\*Bloklashni oldini oladi\*\*: Sinxron dasturlarda biron bir uzun vazifa bajarilsa, dastur bloklanadi va boshqa ishlar bajarilmaydi. Asinxron dasturlashda esa dastur bloklanmasdan ishlashni davom ettiradi.

### \*\*Asinxron dasturlashda eng ko'p ishlatiladigan metodlar:\*\*

1. \*\*Task.Delay\*\*:

- \*\*Task.Delay\*\* asinxron kutish uchun ishlatiladi. Bu metod kutish vaqtini simulyatsiya qilish yoki vaqtinchalik to'xtatish uchun foydalidir.

\*\*Misol\*\*:

```csharp

public async Task TestDelayAsync()

{

Console.WriteLine("Before Delay");

await Task.Delay(2000); // 2 sekund kutish

Console.WriteLine("After Delay");

}

```

2. \*\*ConfigureAwait\*\*:

- \*\*ConfigureAwait\*\* yordamida asinxron metodni qaytarishdan oldin ma'lum bir kontekstdan foydalanish mumkin. Agar \*\*ConfigureAwait(false)\*\* ishlatilsa, UI yoki main thread (bloklanmagan) qayta ishlatiladi, bu esa ko'proq samaradorlikka olib keladi.

\*\*Misol\*\*:

```csharp

public async Task<string> GetDataAsync()

{

await Task.Delay(1000).ConfigureAwait(false); // Bloklanmaydi

return "Data received";

}

```

### \*\*Sinxron vs Asinxron:\*\*

- \*\*Sinxron dasturlash\*\*: Dastur ketma-ket ishlaydi. Agar biron bir vazifa uzun davom etsa (masalan, faylni o'qish), boshqa vazifalar bajarilmaydi.

- \*\*Misol\*\*: `Thread.Sleep()` metodini chaqirish sinxron kutishni amalga oshiradi.

- \*\*Asinxron dasturlash\*\*: Dastur bir vaqtning o'zida ko'plab vazifalarni bajarishi mumkin, kutish vaqtida boshqa ishlar amalga oshadi. Asinxron metodlar odatda \*\*await\*\* yordamida chaqiriladi va bajarilishi davomida bloklanmaydi.

### \*\*Asinxron dasturlash misoli:\*\*

```csharp

using System;

using System.Threading.Tasks;

class Program

{

public static async Task Main(string[] args)

{

Console.WriteLine("Before asynchronous task");

await AsyncMethod();

Console.WriteLine("After asynchronous task");

}

public static async Task AsyncMethod()

{

await Task.Delay(2000); // 2 sekund kutish

Console.WriteLine("Inside asynchronous method");

}

}

```

\*\*Chiqarish\*\*:

```

Before asynchronous task

Inside asynchronous method

After asynchronous task

```

### \*\*Asinxron va paralel dasturlash farqlari:\*\*

- \*\*Asinxron dasturlash\*\*: Bitta threadda bir nechta operatsiyalarni amalga oshirishni ta'minlaydi, bu ko'proq resurslarni tejashga yordam beradi.

- \*\*Paralel dasturlash\*\*: Bir nechta threadlarda bir vaqtning o'zida turli vazifalarni bajarishdir. Bu ko'proq samaradorlikni oshiradi, lekin threadlarni boshqarish murakkablashadi.

### \*\*Xulosa:\*\*

Asinxron dasturlash — bu resurslarni samarali boshqarish va dastur samaradorligini oshirish uchun juda muhim yondashuvdir. Asinxron metodlar yordamida kutish vaqtida boshqa vazifalarni bajarish mumkin, bu esa dasturdan yuqori samaradorlikni olishga yordam beradi. Asinxron dasturlashda \*\*async\*\* va \*\*await\*\* kalit so'zlaridan foydalaniladi.

LINQ

\*\*LINQ\*\* (Language Integrated Query) — bu C# va .NET tilida ma'lumotlar manbalarini (masalan, ma'lumotlar bazasi, kolleksiya, fayl tizimi, va boshqalar) oson va samarali tarzda so'rov qilish uchun ishlatiladigan yondashuvdir. LINQ dasturlash tilida so'rovlarni yozishni oddiylashtiradi, shu bilan birga yuqori darajadagi abstraksiya va ma'lumotlarni filtrlash, guruhlash, tartiblash kabi operatsiyalarni soddalashtiradi.

### \*\*LINQ ning Asosiy Tushunchalari:\*\*

1. \*\*LINQ to Objects\*\*:

- Bu xususiyat yordamida kolleksiya (masalan, massiv yoki ro'yxat) ustida so'rovlar (queries) bajarish mumkin.

\*\*Misol\*\*:

```csharp

List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

var evenNumbers = from num in numbers

where num % 2 == 0

select num;

foreach (var num in evenNumbers)

{

Console.WriteLine(num);

}

```

2. \*\*LINQ to SQL\*\*:

- \*\*LINQ to SQL\*\* yordamida SQL ma'lumotlar bazasiga so'rovlar yuborish mumkin. Bu LINQ va SQL tilini birlashtiradi.

\*\*Misol\*\*:

```csharp

var query = from product in dbContext.Products

where product.Price > 100

select product;

```

3. \*\*LINQ to Entities\*\*:

- \*\*LINQ to Entities\*\* — bu Entity Framework yordamida ma'lumotlar bazasiga LINQ so'rovlarini yuborish imkonini beradi.

4. \*\*LINQ to XML\*\*:

- \*\*LINQ to XML\*\* XML hujjatlari bilan ishlash uchun LINQ ni ishlatishga imkon beradi. XML faylni o'qish, tahrirlash va yaratishda samarali ishlaydi.

5. \*\*LINQ to DataSet\*\*:

- \*\*LINQ to DataSet\*\* yordamida ma'lumotlarni DataSet obyektlari bilan ishlashda LINQ ni qo'llash mumkin.

### \*\*LINQ Asosiy Operatorlari:\*\*

LINQ da ishlatiladigan ba'zi asosiy operatorlar:

1. \*\*`from`\*\*: LINQ so'rovining boshlanishi. Bu operator kolleksiyadagi elementlar ustida so'rovni boshlash uchun ishlatiladi.

\*\*Misol\*\*:

```csharp

var query = from person in people

select person;

```

2. \*\*`where`\*\*: Ma'lumotlar ustida filtrlash qilish uchun ishlatiladi.

\*\*Misol\*\*:

```csharp

var adults = from person in people

where person.Age >= 18

select person;

```

3. \*\*`select`\*\*: Ma'lumotni tanlash, ya'ni, qaysi atributlarni qaytarish kerakligini belgilash.

\*\*Misol\*\*:

```csharp

var names = from person in people

select person.Name;

```

4. \*\*`order by`\*\*: Ma'lumotlarni tartiblash.

\*\*Misol\*\*:

```csharp

var sortedPeople = from person in people

orderby person.Age descending

select person;

```

5. \*\*`group by`\*\*: Ma'lumotlarni guruhlash.

\*\*Misol\*\*:

```csharp

var groupedByAge = from person in people

group person by person.Age into ageGroup

select ageGroup;

```

6. \*\*`join`\*\*: Ikkita ma'lumotlar manbasini birlashtirish.

\*\*Misol\*\*:

```csharp

var query = from order in orders

join customer in customers on order.CustomerId equals customer.CustomerId

select new { order.OrderId, customer.Name };

```

7. \*\*`let`\*\*: O'zgaruvchi yaratish va qiymatlarni saqlash.

\*\*Misol\*\*:

```csharp

var query = from product in products

let discountedPrice = product.Price \* 0.9

select new { product.Name, discountedPrice };

```

8. \*\*`distinct`\*\*: Takrorlanayotgan elementlarni olib tashlash.

\*\*Misol\*\*:

```csharp

var distinctNames = people.Select(p => p.Name).Distinct();

```

9. \*\*`take` va `skip`\*\*: Ma'lumotlarning ma'lum bir qismini olish.

\*\*Misol\*\*:

```csharp

var top3 = people.Take(3); // Eng yuqori 3 ta element

var skip3 = people.Skip(3); // 3 ta birinchi elementni o'tkazib yuborish

```

### \*\*LINQ Extension Methodlari:\*\*

LINQ ning ko'p operatorlari \*\*extension method\*\* (kengaytirilgan metodlar) sifatida mavjud bo'lib, bu metodlar \*\*`IEnumerable<T>`\*\* yoki \*\*`IQueryable<T>`\*\* interfeyslari yordamida ishlaydi.

- \*\*`Where`\*\*: Filtrlash

- \*\*`Select`\*\*: Ma'lumotni tanlash

- \*\*`OrderBy` va `OrderByDescending`\*\*: Tartiblash

- \*\*`GroupBy`\*\*: Guruhlash

- \*\*`Sum`, `Average`, `Min`, `Max`\*\*: Agregat operatsiyalar

- \*\*`First`, `FirstOrDefault`, `Single`, `SingleOrDefault`\*\*: Birinchi yoki yagona elementni olish

- \*\*`Any`, `All`\*\*: To'liq tekshiruv

\*\*Misol\*\*:

```csharp

var names = people.Where(p => p.Age > 18).Select(p => p.Name);

```

### \*\*LINQ va Lambda Expressions\*\*:

LINQ so'rovlari ko'pincha \*\*lambda expressions\*\* bilan birgalikda ishlatiladi. Lambda ifodalari metodlarga qisqa va samarali tarzda argumentlarni uzatishga yordam beradi.

\*\*Misol\*\*:

```csharp

var adults = people.Where(p => p.Age >= 18).ToList();

```

### \*\*LINQ Query va Extension Method Farqi\*\*:

- \*\*LINQ Query\*\* sintaksisi SQL so'rovlariga o'xshash va ko'pincha o'qilishi oson.

- \*\*Extension Methods\*\* sintaksisi metodlar yordamida ishlaydi va ba'zan ko'proq moslashuvchan.

\*\*Misol\*\*:

```csharp

// LINQ Query sintaksisi

var query = from p in people

where p.Age >= 18

select p;

// Extension Method sintaksisi

var query = people.Where(p => p.Age >= 18);

```

### \*\*LINQ ning Foydalari\*\*:

- \*\*Kodning soddaligi\*\*: LINQ yordamida murakkab so'rovlar juda qisqa va aniq yozilishi mumkin.

- \*\*Kengaytirilgan imkoniyatlar\*\*: LINQ nafaqat koleksiya bilan ishlashni osonlashtiradi, balki ma'lumotlar bazasi (LINQ to SQL), XML (LINQ to XML), va boshqa tizimlarga ham ishlov berish imkoniyatini beradi.

- \*\*Xatolarni kamaytirish\*\*: LINQ so'rovlari kompilyatsiya paytida tekshiriladi, bu esa xatolarni dastur ishlashidan oldin aniqlash imkonini beradi.

### \*\*Xulosa\*\*:

LINQ — bu C# tilida ma'lumotlar ustida so'rovlar qilishni soddalashtiruvchi kuchli vosita. LINQ nafaqat koleksiya yoki massivlar bilan ishlashni osonlashtiradi, balki ma'lumotlar bazasi, XML, va boshqa ma'lumot manbalariga so'rovlar yuborish uchun ham ishlatiladi. Bu tizimni samarali va tez ishlashini ta'minlaydi.

SOLID

SOLID\*\* — bu ob'ektga yo'naltirilgan dasturlash (OOP) da kodni mustahkam, moslashuvchan va kengaytiriladigan qilib yozish uchun qo'llaniladigan beshta asosiy tamoyil majmui. SOLID tamoyillarini o'rgangan holda dasturchilar katta va murakkab loyihalarni boshqarishda soddalashtirish va kodni saqlash osonroq bo'lishiga erishishadi.

\*\*SOLID\*\* qisqartmasi quyidagi beshta tamoyilning bosh harflaridan tashkil topgan:

1. \*\*S — Single Responsibility Principle (SRP)\*\*

\*\*Bir vazifa tamoyili\*\*: Har bir sinf yoki modul faqat bitta vazifa uchun javobgar bo'lishi kerak. Agar sinfda bir nechta mas'uliyat bo'lsa, ularni alohida sinflarga bo'lish kerak. Bu kodni tushunish, test qilish va o'zgartirishni osonlashtiradi.

\*\*Misol:\*\* Agar sizda "Employee" (Xodim) sinfi bo'lsa, uning vazifasi faqat xodim ma'lumotlarini boshqarish bo'lishi kerak. Agar ushbu sinfda xodimning maoshi yoki soliqlarini hisoblash uchun kodlar ham mavjud bo'lsa, unda bu kodlarni alohida sinflarga ajratish kerak.

2. \*\*O — Open/Closed Principle (OCP)\*\*

\*\*Ochiq/Yopiq tamoyili\*\*: Kodni kengaytirishga ochiq, ammo o'zgartirishga yopiq qilish kerak. Bu tamoyil kodni qayta yozish o'rniga, yangi funksiyalar qo'shib kengaytirishga imkon beradi.

\*\*Misol:\*\* Tasavvur qiling, "Notification" (Xabar) sinfingiz bor va u faqat e-mail jo'natish funksiyasini bajaradi. Agar siz SMS orqali xabar jo'natishni qo'shishni xohlasangiz, mavjud kodni o'zgartirish o'rniga, yangi sinf yaratib, undan foydalanishingiz kerak.

3. \*\*L — Liskov Substitution Principle (LSP)\*\*

\*\*Liskov almashtirish tamoyili\*\*: Agar `B` sinfi `A` sinfidan meros olsa, `B` sinfini `A` sinfi kabi ishlata olish kerak. Ya'ni, `B` sinfi `A` sinfi o'rnini bosishda dasturda hech qanday xatolik yoki kutilmagan muammolarni yuzaga keltirmasligi kerak.

\*\*Misol:\*\* Agar sizda "Bird" (Qush) sinfi va undan meros olgan "Penguin" (Pingvin) sinfi bo'lsa, "Bird" sinfining uchish funksiyasi mavjud va "Penguin" sinfi bu funksiyani bajara olmaydi. Bu tamoyilni buzadi. Bunday holatda, uchish xususiyatini alohida sinfga chiqarish maqsadga muvofiq bo'ladi.

4. \*\*I — Interface Segregation Principle (ISP)\*\*

\*\*Interfeysni ajratish tamoyili\*\*: Sinflar o'zlari foydalanmaydigan metodlarga majbur bo'lmasligi kerak. Katta interfeyslarni kichik, aniq maqsadga yo'naltirilgan interfeyslarga bo'lish kerak.

\*\*Misol:\*\* Agar sizda "Printer" interfeysi bo'lsa va u print(), scan(), fax() funksiyalariga ega bo'lsa, "OldPrinter" sinfi faqat print() funksiyasidan foydalanadi, lekin scan() va fax() ni qo'llab-quvvatlamaydi. Bu holda interfeyslarni `Printable`, `Scannable`, `Faxable` kabi kichik interfeyslarga bo'lish kerak.

5. \*\*D — Dependency Inversion Principle (DIP)\*\*

\*\*Bog'lanishning teskari tamoyili\*\*: Yuqori darajadagi modullar past darajadagi modullarga bog'liq bo'lmasligi kerak. Ikkalasi ham abstraktsiyalarga bog'liq bo'lishi kerak, ya'ni sinf o'z ichidagi konkret sinflarga bog'lanib qolmasligi kerak, aksincha interfeyslar yoki abstrakt sinflar orqali ishlashi kerak.

\*\*Misol:\*\* Agar "OrderService" (Buyurtma Xizmati) sinfi "EmailService" sinfiga bog'langan bo'lsa, "OrderService" faqat e-mail orqali xabar jo'nata oladi. Agar siz boshqa xabar jo'natish usulini qo'shmoqchi bo'lsangiz, ushbu bog'liqlik DIP ni buzadi. Buning o'rniga, umumiy `NotificationService` interfeysi yaratib, "OrderService" uni qabul qilishi kerak.

---

\*\*SOLID tamoyillarining afzalliklari:\*\*

- Kodni \*\*qayta ishlash\*\* va \*\*kengaytirish\*\* osonlashadi.

- \*\*Qayta foydalanish\*\* imkoniyatini oshiradi.

- \*\*Testlashni\*\* soddalashtiradi va kodni \*\*o'zgartirishni\*\* kamroq murakkab qiladi.

\*\*Umumiy xulosa:\*\* SOLID tamoyillari dasturiy ta'minotni mustahkam va ishonchli qiladi, kodni saqlash va kengaytirishni osonlashtiradi. Bu tamoyillar dasturiy ta'minotni loyihalashda muhim qoidalar bo'lib, ulardan foydalangan holda kodni strukturaviy jihatdan yaxshi holatda saqlash mumkin.