Интерфейси — 12б, ООП

1. Увод

Интерфейсите са контракти, които дефинират какво трябва да прави един клас, но не как да го прави. Те позволяват множествено наследяване и осигуряват гъвкавост в дизайна на програмите.

2. Какво е интерфейс?

Интерфейсите са:

- **Контракти** - дефинират какво трябва да прави класът
- **Абстрактни** - не съдържат имплементация
- **Множествено наследяване** - клас може да имплементира много интерфейси
- **Публични** - всички членове са публични
- **Принудителни** - класът трябва да имплементира всички методи

3. Синтаксис на интерфейсите

Основен синтаксис:

- **interface ключова дума** - дефинира интерфейс
- **I префикс** - конвенция за именуване

- **Методи без имплементация** - само декларация
- **Свойства** - могат да се дефинират
- **Събития** - могат да се дефинират

Основен пример с интерфейс:

```
// Дефиниране на интерфейс
interface IShape {
    double CalculateArea();
    double CalculatePerimeter();
    void DisplayInfo();
}

// Имплементация на интерфейса
class Rectangle : IShape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    // Имплементация на методите от интерфейса
    public double CalculateArea() {
        return width * height;
    }

    public double CalculatePerimeter() {
        return 2 * (width + height);
    }

    public void DisplayInfo() {
        Console.WriteLine($"Правоъгълник: {width}x{height}");
        Console.WriteLine($"Площ: {CalculateArea():F2}");
        Console.WriteLine($"Периметър: {CalculatePerimeter():F2}");
    }
}

class Circle : IShape {
    private double radius;
```

```
    public Circle(double radius) {
        this.radius = radius;
    }

    public double CalculateArea() {
        return Math.PI * radius * radius;
    }

    public double CalculatePerimeter() {
        return 2 * Math.PI * radius;
    }

    public void DisplayInfo() {
        Console.WriteLine($"Кръг с радиус: {radius}");
        Console.WriteLine($"Площ: {CalculateArea():F2}");
        Console.WriteLine($"Периметър: {CalculatePerimeter():F2}");
    }
}

// Използване
IShape shape1 = new Rectangle(5, 3);
IShape shape2 = new Circle(4);

shape1.DisplayInfo();
shape2.DisplayInfo();
```

4. Множествено наследяване с интерфейси

Предимства на множественото наследяване:

- **Гъвкавост** - клас може да има различни способности
- **Модулност** - разделяне на функционалността
- **Преизползване** - интерфейсите могат да се комбинират
- **Стандартизация** - дефиниране на общи контракти

Пример с множествено наследяване:

```
// Интерфейс за летене
interface IFlyable {
    void Fly();
    double GetMaxAltitude();
}

// Интерфейс за плуване
interface ISwimmable {
    void Swim();
    double GetMaxDepth();
}

// Интерфейс за ходене
interface IWalkable {
    void Walk();
    double GetMaxSpeed();
}

// Интерфейс за звуци
interface ISoundable {
    void MakeSound();
    string GetSoundType();
}

// Клас, който имплементира множество интерфейси
class Duck : IFlyable, ISwimmable, IWalkable, ISoundable {
    private string name;
    private double maxAltitude;
    private double maxDepth;
    private double maxSpeed;

    public Duck(string name) {
        this.name = name;
        this.maxAltitude = 1000; // метра
        this.maxDepth = 5; // метра
        this.maxSpeed = 10; // км/ч
    }

    // Имплементация на IFlyable
    public void Fly() {
        Console.WriteLine($"{name} лети в небето");
    }

    public double GetMaxAltitude() {
        return maxAltitude;
    }
```

```
    // Имплементация на ISwimmable
    public void Swim() {
        Console.WriteLine($"{name} плува в езерото");
    }

    public double GetMaxDepth() {
        return maxDepth;
    }

    // Имплементация на IWalkable
    public void Walk() {
        Console.WriteLine($"{name} ходи по земята");
    }

    public double GetMaxSpeed() {
        return maxSpeed;
    }

    // Имплементация на ISoundable
    public void MakeSound() {
        Console.WriteLine($"{name} кряка: Quack! Quack!");
    }

    public string GetSoundType() {
        return "Крякане";
    }

    // Допълнителен метод
    public void DisplayAbilities() {
        Console.WriteLine($"Патокът {name} може да:");
        Console.WriteLine($"- Лети до {maxAltitude}м височина");
        Console.WriteLine($"- Плува до {maxDepth}м дълбочина");
        Console.WriteLine($"- Ходи със скорост {maxSpeed} км/ч");
        Console.WriteLine($"- Прави звук: {GetSoundType()}");
    }
}

// Използване
Duck duck = new Duck("Доналд");

// Използване чрез различни интерфейси
IFlyable flyable = duck;
ISwimmable swimmable = duck;
IWalkable walkable = duck;
ISoundable soundable = duck;
```

```
flyable.Fly();
swimmable.Swim();
walkable.Walk();
soundable.MakeSound();

duck.DisplayAbilities();
```

5. Свойства и събития в интерфейси

Интерфейсите могат да съдържат:

- **Свойства** - get и set аксесори
- **Събития** - за комуникация между обекти
- **Индекси** - за достъп като масиви
- **Методи** - основната функционалност

Пример с свойства и събития:

```
// Интерфейс за уведомления
interface INotifiable {
    string Name { get; set; }
    string Email { get; set; }
    bool IsActive { get; set; }

    event EventHandler NotificationReceived;

    void SendNotification(string message);
    void Subscribe();
    void Unsubscribe();
}

// Интерфейс за валидация
interface IValidatable {
    bool IsValid { get; }
    List ValidationErrors { get; }
```

```
    bool Validate();
    void ClearErrors();
}

// Интерфейс за сериализация
interface ISerializable {
    string Serialize();
    void Deserialize(string data);
    string FileExtension { get; }
}

// Клас, който имплементира множество интерфейси
class User : INotifiable, IValidatable, ISerializable {
    private string name;
    private string email;
    private bool isActive;
    private List validationErrors;

    public event EventHandler NotificationReceived;

    public User(string name, string email) {
        this.name = name;
        this.email = email;
        this.isActive = true;
        this.validationErrors = new List();
    }

    // Имплементация на INotifiable
    public string Name {
        get { return name; }
        set { name = value; }
    }

    public string Email {
        get { return email; }
        set { email = value; }
    }

    public bool IsActive {
        get { return isActive; }
        set { isActive = value; }
    }

    public void SendNotification(string message) {
        if (isActive) {
            Console.WriteLine($"Изпратено уведомление до {name} ({email}):
{message}");
```

```
            NotificationReceived?.Invoke(this, message);
        }
    }

    public void Subscribe() {
        Console.WriteLine($"{name} се абонира за уведомления");
    }

    public void Unsubscribe() {
        Console.WriteLine($"{name} се отписва от уведомления");
        isActive = false;
    }

    // Имплементация на IValidatable
    public bool IsValid {
        get { return validationErrors.Count == 0; }
    }

    public List ValidationErrors {
        get { return new List(validationErrors); }
    }

    public bool Validate() {
        validationErrors.Clear();

        if (string.IsNullOrEmpty(name)) {
            validationErrors.Add("Името не може да бъде празно");
        }

        if (string.IsNullOrEmpty(email) || !email.Contains("@")) {
            validationErrors.Add("Невалиден email адрес");
        }

        return IsValid;
    }

    public void ClearErrors() {
        validationErrors.Clear();
    }

    // Имплементация на ISerializable
    public string Serialize() {
        return $"Name:{name};Email:{email};Active:{isActive}";
    }

    public void Deserialize(string data) {
        var parts = data.Split(';');
```

```
        foreach (var part in parts) {
            var keyValue = part.Split(':');
            if (keyValue.Length == 2) {
                switch (keyValue[0]) {
                    case "Name":
                        name = keyValue[1];
                        break;
                    case "Email":
                        email = keyValue[1];
                        break;
                    case "Active":
                        isActive = bool.Parse(keyValue[1]);
                        break;
                }
            }
        }
    }

    public string FileExtension {
        get { return ".user"; }
    }
}

// Използване
User user = new User("Иван Петров", "ivan@example.com");

// Валидация
if (user.Validate()) {
    Console.WriteLine("Потребителят е валиден");
} else {
    Console.WriteLine("Грешки при валидация:");
    foreach (var error in user.ValidationErrors) {
        Console.WriteLine($"- {error}");
    }
}

// Уведомления
user.NotificationReceived += (sender, message) => {
    Console.WriteLine($"Получено уведомление: {message}");
};

user.SendNotification("Добре дошли в системата!");

// Сериализация
string serialized = user.Serialize();
Console.WriteLine($"Сериализиран потребител: {serialized}");
```

```
User newUser = new User("", "");
newUser.Deserialize(serialized);
Console.WriteLine($"Възстановен потребител: {newUser.Name}");
```

6. Наследяване на интерфейси

Интерфейсите могат да се наследяват:

- **От други интерфейси** - наследяване на методи
- **Множествено наследяване** - интерфейс може да наследява множество интерфейси
- **Разширяване** - добавяне на нови методи
- **Комбиниране** - създаване на сложни контракти

Пример с наследяване на интерфейси:

```
// Базов интерфейс за животни
interface IAnimal {
    string Name { get; set; }
    int Age { get; set; }
    void Eat();
    void Sleep();
}

// Интерфейс за домашни животни
interface IPet : IAnimal {
    string Owner { get; set; }
    void Play();
    void BePetted();
}

// Интерфейс за диви животни
interface IWildAnimal : IAnimal {
    string Habitat { get; set; }
    bool IsDangerous { get; }
```

```
    void Hunt();
}

// Интерфейс за летящи животни
interface IFlyingAnimal : IAnimal {
    double MaxAltitude { get; }
    void Fly();
}

// Интерфейс за плуващи животни
interface ISwimmingAnimal : IAnimal {
    double MaxDepth { get; }
    void Swim();
}

// Комбиниран интерфейс за водоплаващи птици
interface IWaterfowl : IFlyingAnimal, ISwimmingAnimal, IWildAnimal {
    string MigrationRoute { get; set; }
    void Migrate();
}

// Имплементация на сложния интерфейс
class Duck : IWaterfowl {
    public string Name { get; set; }
    public int Age { get; set; }
    public string Habitat { get; set; }
    public string MigrationRoute { get; set; }
    public double MaxAltitude { get; private set; }
    public double MaxDepth { get; private set; }

    public bool IsDangerous => false;

    public Duck(string name, int age) {
        Name = name;
        Age = age;
        Habitat = "Езеро";
        MigrationRoute = "Север-Юг";
        MaxAltitude = 1000;
        MaxDepth = 5;
    }

    // Имплементация на IAnimal
    public void Eat() {
        Console.WriteLine($"{Name} яде водни растения и малки рибки");
    }

    public void Sleep() {
```

```
        Console.WriteLine($"{Name} спи на водата");
    }

    // Имплементация на IWildAnimal
    public void Hunt() {
        Console.WriteLine($"{Name} лови малки рибки");
    }

    // Имплементация на IFlyingAnimal
    public void Fly() {
        Console.WriteLine($"{Name} лети до {MaxAltitude}м височина");
    }

    // Имплементация на ISwimmingAnimal
    public void Swim() {
        Console.WriteLine($"{Name} плува до {MaxDepth}м дълбочина");
    }

    // Имплементация на IWaterfowl
    public void Migrate() {
        Console.WriteLine($"{Name} мигрира по маршрута: {MigrationRoute}");
    }

    // Допълнителен метод
    public void DisplayInfo() {
        Console.WriteLine($"Патък: {Name}, {Age} години");
        Console.WriteLine($"Обиталище: {Habitat}");
        Console.WriteLine($"Миграционен маршрут: {MigrationRoute}");
        Console.WriteLine($"Максимална височина: {MaxAltitude}м");
        Console.WriteLine($"Максимална дълбочина: {MaxDepth}м");
        Console.WriteLine($"Опасен: {(IsDangerous ? "Да" : "Не")}");
    }
}

// Използване
Duck duck = new Duck("Доналд", 3);

// Използване чрез различни интерфейси
IAnimal animal = duck;
IWildAnimal wildAnimal = duck;
IFlyingAnimal flyingAnimal = duck;
ISwimmingAnimal swimmingAnimal = duck;
IWaterfowl waterfowl = duck;

animal.Eat();
wildAnimal.Hunt();
flyingAnimal.Fly();
```

```
swimmingAnimal.Swim();
waterfowl.Migrate();

duck.DisplayInfo();
```

7. Пълен пример - Система за плащания

Реален пример с множество интерфейси:

```
// Интерфейс за плащания
interface IPayment {
    decimal Amount { get; set; }
    string Currency { get; set; }
    DateTime PaymentDate { get; set; }
    bool ProcessPayment();
    string GetPaymentDetails();
}

// Интерфейс за валидация
interface IValidatable {
    bool IsValid { get; }
    List ValidationErrors { get; }
    bool Validate();
}

// Интерфейс за логиране
interface ILoggable {
    void Log(string message);
    string GetLogInfo();
}

// Интерфейс за нотификации
interface INotifiable {
    event EventHandler PaymentProcessed;
    void SendNotification(string message);
}

// Интерфейс за рефундиране
interface IRefundable {
    bool CanRefund { get; }
    bool ProcessRefund();
```

```
    string GetRefundDetails();
}

// Интерфейс за криптиране
interface IEncryptable {
    string Encrypt(string data);
    string Decrypt(string encryptedData);
}

// Базов клас за плащания
public abstract class PaymentBase : IPayment, IValidatable, ILoggable,
INotifiable {
    public decimal Amount { get; set; }
    public string Currency { get; set; }
    public DateTime PaymentDate { get; set; }
    public bool IsValid { get; private set; }
    public List ValidationErrors { get; private set; }

    public event EventHandler PaymentProcessed;

    protected PaymentBase(decimal amount, string currency) {
        Amount = amount;
        Currency = currency;
        PaymentDate = DateTime.Now;
        ValidationErrors = new List();
        IsValid = false;
    }

    // Имплементация на IPayment
    public abstract bool ProcessPayment();

    public virtual string GetPaymentDetails() {
        return $"Плащане: {Amount:C} {Currency} на {PaymentDate:dd.MM.yyyy
HH:mm}";
    }

    // Имплементация на IValidatable
    public virtual bool Validate() {
        ValidationErrors.Clear();

        if (Amount <= 0) {
            ValidationErrors.Add("Сумата трябва да е положителна");
        }

        if (string.IsNullOrEmpty(Currency)) {
            ValidationErrors.Add("Валутата не може да бъде празна");
        }
```

```
        if (PaymentDate > DateTime.Now) {
            ValidationErrors.Add("Датата на плащането не може да бъде в
бъдещето");
        }

        IsValid = ValidationErrors.Count == 0;
        return IsValid;
    }

    // Имплементация на ILoggable
    public virtual void Log(string message) {
        Console.WriteLine($"[{DateTime.Now:HH:mm:ss}] {GetType().Name}:
{message}");
    }

    public virtual string GetLogInfo() {
        return $"{GetType().Name} - {Amount:C} {Currency}";
    }

    // Имплементация на INotifiable
    public virtual void SendNotification(string message) {
        Log($"Изпратено уведомление: {message}");
        PaymentProcessed?.Invoke(this, message);
    }

    // Защитен метод за обща логика
    protected virtual void OnPaymentProcessed() {
        Log("Плащането е обработено успешно");
        SendNotification($"Плащането от {Amount:C} е завършено");
    }
}

// Конкретна имплементация за кредитна карта
public class CreditCardPayment : PaymentBase, IRefundable, IEncryptable {
    public string CardNumber { get; set; }
    public string CardHolderName { get; set; }
    public DateTime ExpiryDate { get; set; }
    public string CVV { get; set; }

    public bool CanRefund => PaymentDate.AddDays(30) > DateTime.Now;

    public CreditCardPayment(decimal amount, string currency, string
cardNumber,
                             string cardHolderName, DateTime expiryDate,
string cvv)
                             : base(amount, currency) {
```

```
        CardNumber = cardNumber;
        CardHolderName = cardHolderName;
        ExpiryDate = expiryDate;
        CVV = cvv;
    }

    public override bool ProcessPayment() {
        if (!Validate()) {
            Log("Плащането не е валидно");
            return false;
        }

        Log("Обработва се плащане с кредитна карта...");

        // Симулация на обработка
        if (SimulatePaymentProcessing()) {
            OnPaymentProcessed();
            return true;
        }

        Log("Плащането е неуспешно");
        return false;
    }

    public override bool Validate() {
        bool baseValid = base.Validate();

        if (string.IsNullOrEmpty(CardNumber) || CardNumber.Length != 16) {
            ValidationErrors.Add("Невалиден номер на карта");
        }

        if (string.IsNullOrEmpty(CardHolderName)) {
            ValidationErrors.Add("Името на притежателя не може да бъде
празно");
        }

        if (ExpiryDate < DateTime.Now) {
            ValidationErrors.Add("Картата е изтекла");
        }

        if (string.IsNullOrEmpty(CVV) || CVV.Length != 3) {
            ValidationErrors.Add("Невалиден CVV код");
        }

        IsValid = baseValid && ValidationErrors.Count == 0;
        return IsValid;
    }
```

```
    public override string GetPaymentDetails() {
        return base.GetPaymentDetails() +
               $"\nКарта: {MaskCardNumber(CardNumber)}" +
               $"\nПритежател: {CardHolderName}" +
               $"\nИзтича: {ExpiryDate:MM/yyyy}";
    }

    // Имплементация на IRefundable
    public bool ProcessRefund() {
        if (!CanRefund) {
            Log("Рефундът не е възможен - изтекла е 30-дневната граница");
            return false;
        }

        Log($"Обработва се рефунд от {Amount:C}");
        SendNotification($"Рефунд от {Amount:C} е обработен");
        return true;
    }

    public string GetRefundDetails() {
        return $"Рефунд: {Amount:C} {Currency} - {(CanRefund ? "Възможен" :
"Не е възможен")}";
    }

    // Имплементация на IEncryptable
    public string Encrypt(string data) {
        // Проста симулация на криптиране
        return
Convert.ToBase64String(System.Text.Encoding.UTF8.GetBytes(data));
    }

    public string Decrypt(string encryptedData) {
        // Проста симулация на декриптиране
        return
System.Text.Encoding.UTF8.GetString(Convert.FromBase64String(encryptedData)
);
    }

    // Приватни помощни методи
    private bool SimulatePaymentProcessing() {
        // Симулация на успешно плащане
        return new Random().Next(1, 10) > 2; // 80% успех
    }

    private string MaskCardNumber(string cardNumber) {
        if (cardNumber.Length < 8) return "****";
```

```
        return cardNumber.Substring(0, 4) + " **** **** " +
cardNumber.Substring(cardNumber.Length - 4);
    }
}

// Конкретна имплементация за банков превод
public class BankTransferPayment : PaymentBase, IRefundable {
    public string BankName { get; set; }
    public string AccountNumber { get; set; }
    public string RoutingNumber { get; set; }

    public bool CanRefund => PaymentDate.AddDays(7) > DateTime.Now;

    public BankTransferPayment(decimal amount, string currency, string
bankName,
                               string accountNumber, string routingNumber)
                               : base(amount, currency) {
        BankName = bankName;
        AccountNumber = accountNumber;
        RoutingNumber = routingNumber;
    }

    public override bool ProcessPayment() {
        if (!Validate()) {
            Log("Плащането не е валидно");
            return false;
        }

        Log("Обработва се банков превод...");

        // Симулация на обработка
        if (SimulateBankTransfer()) {
            OnPaymentProcessed();
            return true;
        }

        Log("Банковият превод е неуспешен");
        return false;
    }

    public override bool Validate() {
        bool baseValid = base.Validate();

        if (string.IsNullOrEmpty(BankName)) {
            ValidationErrors.Add("Името на банката не може да бъде
празно");
        }
```

```
        if (string.IsNullOrEmpty(AccountNumber) || AccountNumber.Length <
8) {
            ValidationErrors.Add("Невалиден номер на сметка");
        }

        if (string.IsNullOrEmpty(RoutingNumber) || RoutingNumber.Length !=
9) {
            ValidationErrors.Add("Невалиден рутинг номер");
        }

        IsValid = baseValid && ValidationErrors.Count == 0;
        return IsValid;
    }

    public override string GetPaymentDetails() {
        return base.GetPaymentDetails() +
               $"\nБанка: {BankName}" +
               $"\nСметка: {MaskAccountNumber(AccountNumber)}" +
               $"\nРутинг: {RoutingNumber}";
    }

    // Имплементация на IRefundable
    public bool ProcessRefund() {
        if (!CanRefund) {
            Log("Рефундът не е възможен - изтекла е 7-дневната граница");
            return false;
        }

        Log($"Обработва се рефунд от {Amount:C}");
        SendNotification($"Рефунд от {Amount:C} е обработен");
        return true;
    }

    public string GetRefundDetails() {
        return $"Рефунд: {Amount:C} {Currency} - {(CanRefund ? "Възможен" :
"Не е възможен")}";
    }

    // Приватни помощни методи
    private bool SimulateBankTransfer() {
        // Симулация на успешен банков превод
        return new Random().Next(1, 10) > 1; // 90% успех
    }

    private string MaskAccountNumber(string accountNumber) {
        if (accountNumber.Length < 4) return "****";
```

```
        return "****" + accountNumber.Substring(accountNumber.Length - 4);
    }
}

// Използване
var creditCardPayment = new CreditCardPayment(1000, "BGN",
"1234567890123456",
                                              "Иван Петров",
DateTime.Now.AddYears(2), "123");

var bankTransferPayment = new BankTransferPayment(500, "BGN", "Добротворна
банка",
                                                  "1234567890", "123456789");

// Обработка на плащанията
Console.WriteLine("=== Обработка на плащания ===");

if (creditCardPayment.ProcessPayment()) {
    Console.WriteLine("Кредитната карта е обработена успешно");
    Console.WriteLine(creditCardPayment.GetPaymentDetails());
} else {
    Console.WriteLine("Грешки при обработка на кредитната карта:");
    foreach (var error in creditCardPayment.ValidationErrors) {
        Console.WriteLine($"- {error}");
    }
}

Console.WriteLine();

if (bankTransferPayment.ProcessPayment()) {
    Console.WriteLine("Банковият превод е обработен успешно");
    Console.WriteLine(bankTransferPayment.GetPaymentDetails());
} else {
    Console.WriteLine("Грешки при обработка на банковия превод:");
    foreach (var error in bankTransferPayment.ValidationErrors) {
        Console.WriteLine($"- {error}");
    }
}
```

8. Практически задачи

Задачи за упражнение:

- **Създай интерфейс IReadable** с методи за четене на данни
- **Имплементирай интерфейс IWritable** с методи за записване
- **Направи интерфейс IComparable** за сравняване на обекти
- **Създай интерфейс ICloneable** за копиране на обекти
- **Имплементирай интерфейс IDisposable** за освобождаване на ресурси

9. Заключение

Интерфейсите са мощна функционалност, която ни позволява да дефинираме контракти и да осигуряваме гъвкавост в дизайна на програмите чрез множествено наследяване.

Ключови принципи:

- **Контракти** - дефиниране на задължителни методи
- **Множествено наследяване** - клас може да имплементира множество интерфейси
- **Стандартизация** - дефиниране на общи интерфейси
- **Гъвкавост** - лесно добавяне на нови способности
- **Модулност** - разделяне на функционалността