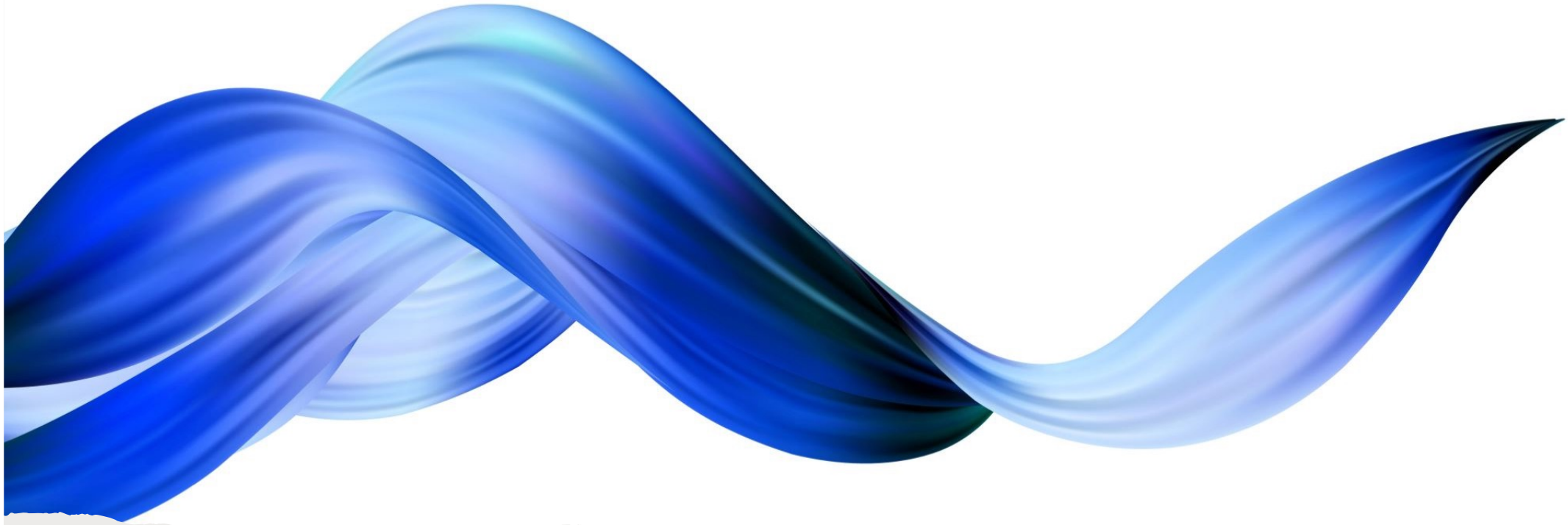


Програмни среди

ТЕОРИЯ

NAMESPACES

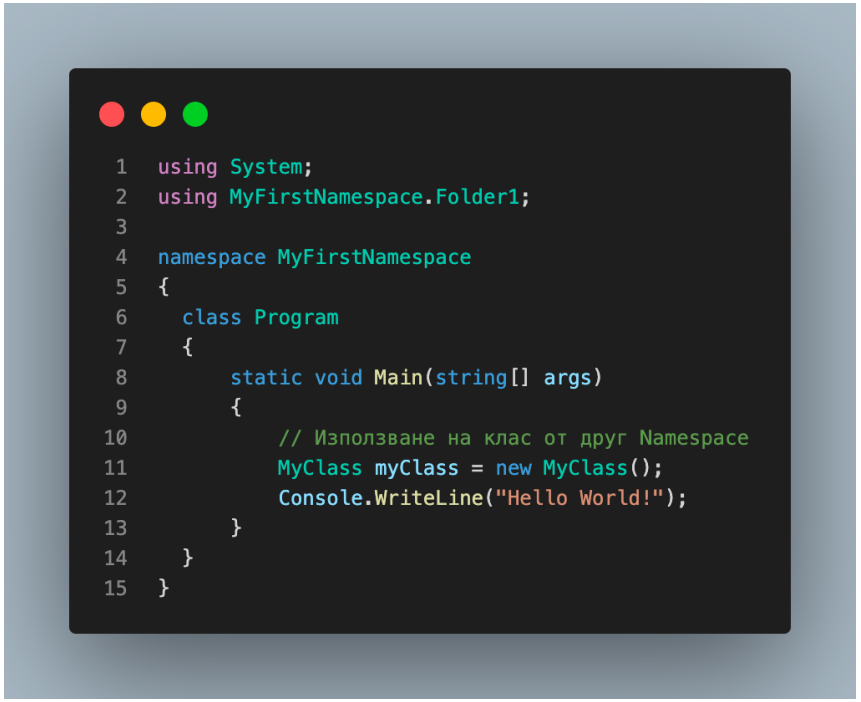
Namespaces в C# са инструмент, който ни позволява да групираме и организираме нашия код. Те ни помагат да избегнем конфликти на имената на типовете, като ги дефинираме в отделни логически групи. Това прави управлението на имената по-лесно и по-организирано при работа с големи проекти. Като цяло, Namespaces са важен елемент от съвременните програмни езици и технологии.

Как да декларираме Namespace в C#?

За да декларираме Namespace в C#, трябва да използваме ключовата дума "namespace", последвана от името на Namespace-а. Може да има повече от един Namespace в един файл.

Как да използваме класове от друг Namespace?

За да използваме класове от друг Namespace, трябва да използваме името на Namespace-а, последвано от точка и името на класа. Може да използваме ключовата дума "using", за да съкратим името на Namespace-а.



```
1  using System;
2  using MyFirstNamespace.Folder1;
3
4  namespace MyFirstNamespace
5  {
6    class Program
7    {
8        static void Main(string[] args)
9        {
10           // Използване на клас от друг Namespace
11           MyClass myClass = new MyClass();
12           Console.WriteLine("Hello World!");
13       }
14   }
15 }
```

ENUM

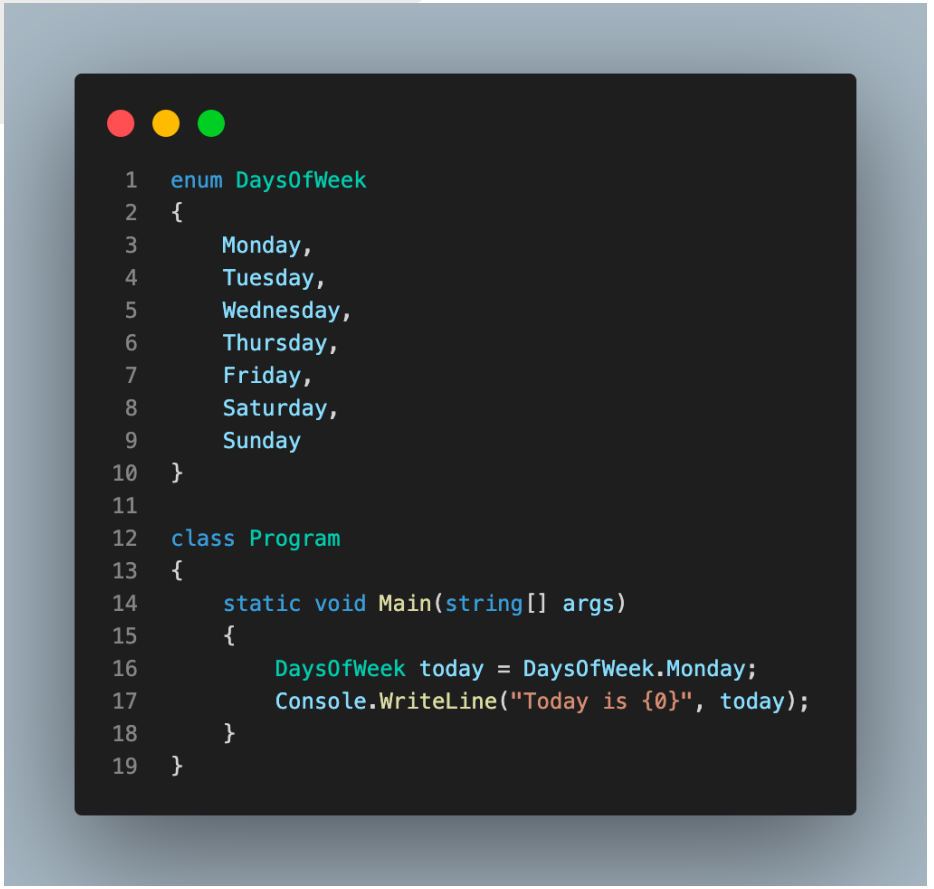
Enum в C# е тип данни, който съдържа списък от константи, които могат да бъдат използвани за заместване на целочислени стойности или константи в програмния код. Enum стойностите са фиксирани и лесно разбираеми, което може да доведе до по-лесно разбиране на програмния код.

Как да декларираме Enum в C#?

За да декларираме enum в C#, трябва да използваме ключовата дума "enum", последвана от името на enum-а и фигурни скоби, които съдържат стойностите.

Как да използваме Enum?

За да използваме enum в програмния код, можем да декларираме променлива с типа на enum и да присвояваме стойности на тази променлива. За да използваме enum, трябва да използваме името на enum-а, последвано от точка и името на стойността.



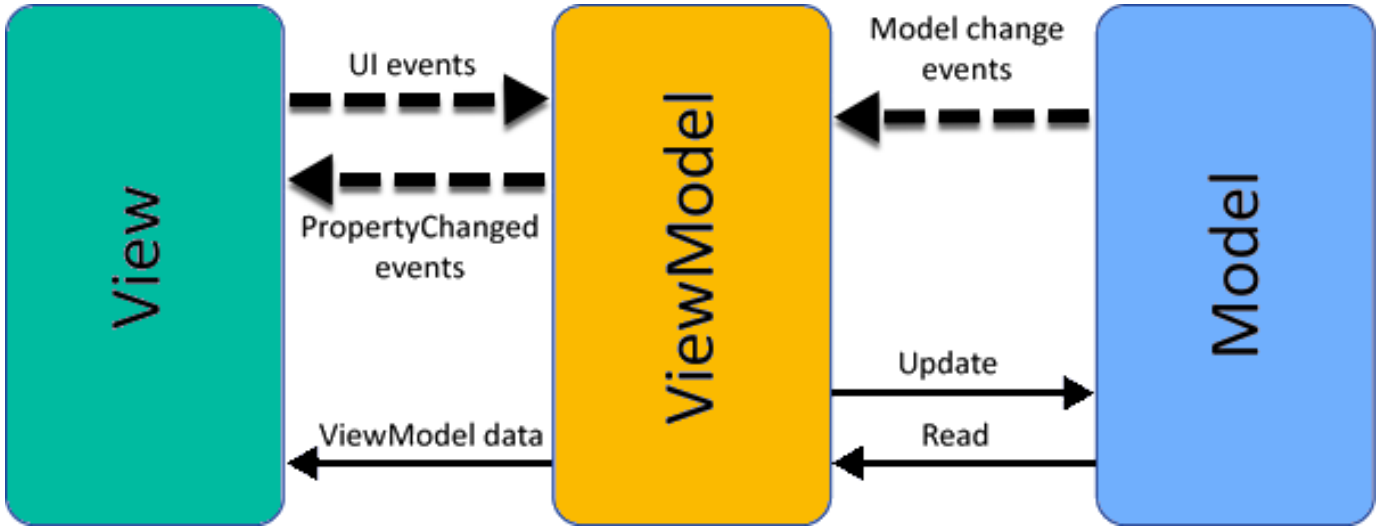
```
1  enum DaysOfWeek
2  {
3      Monday,
4      Tuesday,
5      Wednesday,
6      Thursday,
7      Friday,
8      Saturday,
9      Sunday
10 }
11
12 class Program
13 {
14     static void Main(string[] args)
15     {
16         DaysOfWeek today = DaysOfWeek.Monday;
17         Console.WriteLine("Today is {0}", today);
18     }
19 }
```

MVVM

MVVM (Model-View-ViewModel) е популярен архитектурен шаблон за софтуерни приложения, който се използва в разработката на уеб и десктоп приложения.

Този подход позволява да се раздели логиката на представянето на данните от техния модел и да се постигне по-голяма гъвкавост, тестваемост и поддръжка на кода.

MVVM – ГРАФИЧНО ПРЕДСТАВЯНЕ



```
1  class Person
2  {
3      public string FirstName { get; set; }
4      ...
5  }
```

Model

В MVVM архитектурата моделът (Model) съдържа данни и бизнес логика.

ViewModel

ViewModel служи като медиатор между модела и изгледа, управлявайки данните, които се показват на изгледа. ViewModel-ът съдържа свойства, които представят данните от модела и методи, които позволяват на изгледа да ги манипулира.

```
1  class PersonViewModel
2  {
3      private readonly Person _person;
4
5      public PersonViewModel(Person person)
6      {
7          _person = person;
8      }
9
10     public string FirstName
11     {
12         get => _person.FirstName;
13         set => _person.FirstName = value;
14     }
15
16     ...
17 }
```

View

Изгледът (View) представлява графичния интерфейс на потребителя

```
class PersonView
{
    private readonly PersonViewModel _viewModel;

    public PersonView(PersonViewModel viewModel)
    {
        _viewModel = viewModel;
    }

    public void Display()
    {
        Console.WriteLine("Enter first name:");
        _viewModel.FirstName = Console.ReadLine();

        ...

        _viewModel.Save();

        Console.WriteLine("Data:\n{0}", _viewModel.FirstName);
    }
}
```

```
static void Main(string[] args)
{
    Person person = new Person();
    PersonViewModel viewModel = new PersonViewModel(person);
    PersonView view = new PersonView(viewModel);
    view.Display();
}
```

Main

Навързване на логиката

ПОЛЕ

Поле (field) в C# е променлива, която се дефинира в тялото на класа. Полето може да бъде достъпвано и модифицирано директно от обектите, които са инстанции на този клас.

```
1  class Person
2  {
3      public string Name; // поле Name
4      public int Age; // поле Age
5  }
```

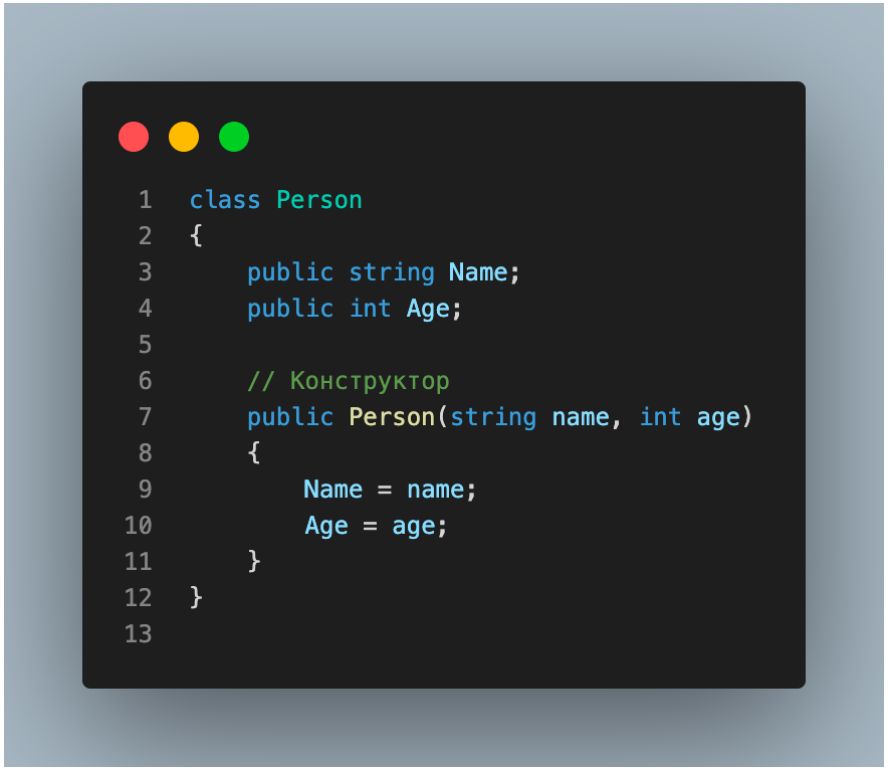
```
1  class Person
2  {
3      private string _name; // поле Name
4
5      public string Name
6      {
7          get { return _name; }
8          set
9          {
10             if (string.IsNullOrEmpty(value))
11                 throw new ArgumentException("Name cannot be null or empty");
12             _name = value;
13         }
14     }
15 }
```

СВОЙСТВО

Свойство (property) в C# предоставя контролиран достъп до полетата на класа. Свойствата позволяват на програмиста да валидира и/или обработва данните, които се записват в полетата, преди да ги върне или да ги запише.

КОНСТРУКТОР

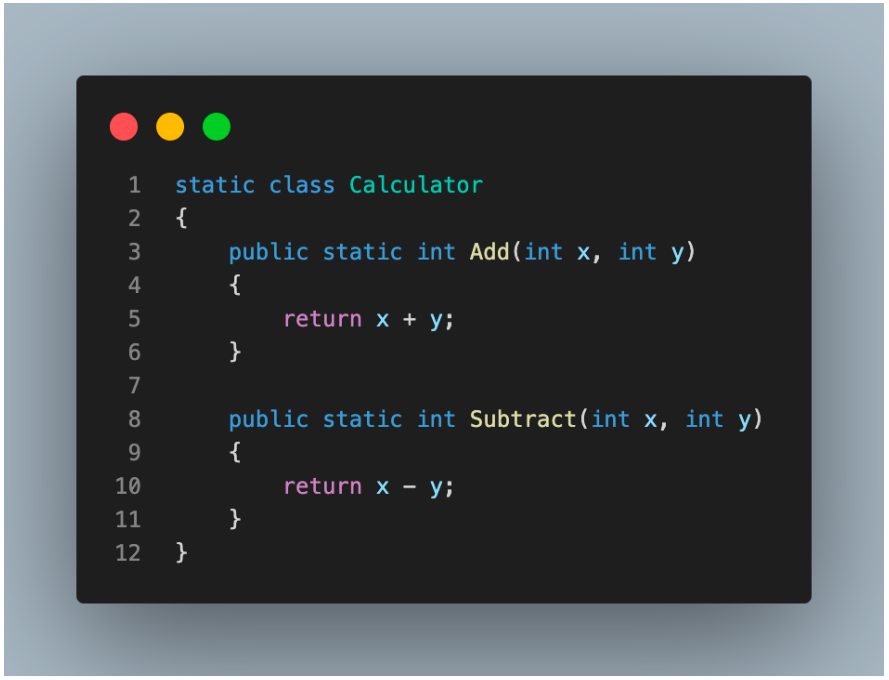
Конструкторът е специален метод в класа, който се използва за инициализиране на обектите. Конструкторът се извиква при създаването на нов обект от класа и може да приема параметри, които се използват за инициализиране на полетата на обекта.



```
1  class Person
2  {
3      public string Name;
4      public int Age;
5
6      // Конструктор
7      public Person(string name, int age)
8      {
9          Name = name;
10         Age = age;
11     }
12 }
13
```

СТАТИЧЕН КЛАС

Статичният клас в C# е клас, който има само статични методи, полета, свойства и събития. Не може да бъде инстанциран, т.е. не могат да се създават обекти от него. Статичният клас се използва за групиране на методи и полета, които са свързани логически, но не изискват съхранение на състояние.



```
1  static class Calculator
2  {
3      public static int Add(int x, int y)
4      {
5          return x + y;
6      }
7
8      public static int Subtract(int x, int y)
9      {
10         return x - y;
11     }
12 }
```

EXCEPTIONS

Exceptions в C# са обекти, които сигнализират за грешки или изключения, които се случват по време на изпълнението на програмата. Когато в програмата се появи изключение, тя се прекъсва, а изключението се хвърля като обект, който може да бъде прихванат и обработен.

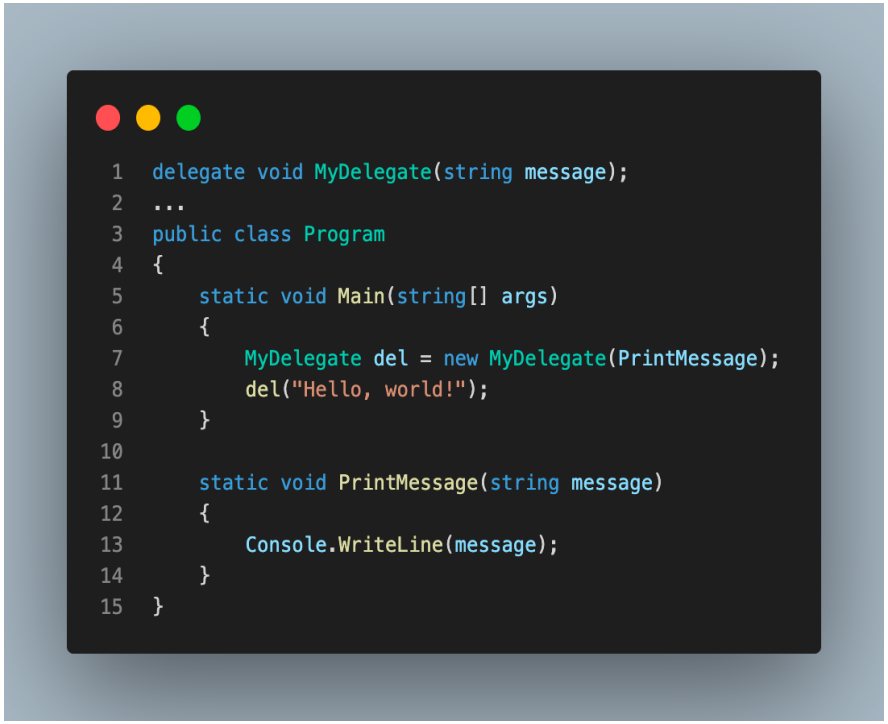
За да прихванем и обработим изключения в C#, използваме блок try-catch. В блока try поставяме код, който може да доведе до генериране на изключение, а в блока catch поставяме код, който се изпълнява, когато изключението се хване.

```
try
{
    int x = int.Parse(Console.ReadLine());
    int y = int.Parse(Console.ReadLine());

    int result = x / y;
    Console.WriteLine("Result: " + result);
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

DELEGATES

Делегатите в C# са типове, които съдържат указател към метод или методи със същата сигнатура като този на делегата. Те позволяват да се подават методи като параметри на други методи, да се връщат методи като резултат от методи, да се използват вместо интерфейси за обратно извикване (callback) и др.



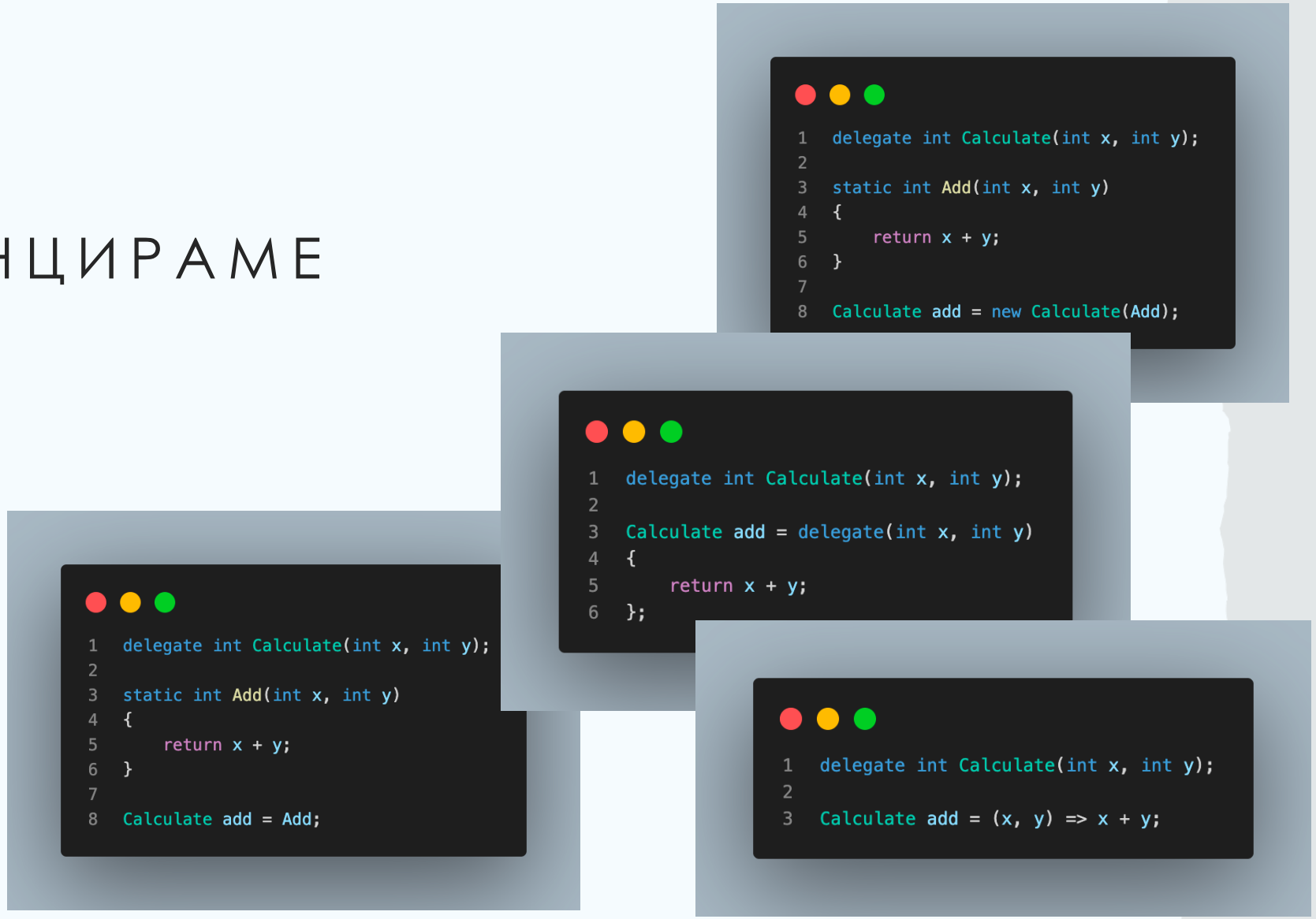
```
1  delegate void MyDelegate(string message);
2  ...
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          MyDelegate del = new MyDelegate(PrintMessage);
8          del("Hello, world!");
9      }
10
11     static void PrintMessage(string message)
12     {
13         Console.WriteLine(message);
14     }
15 }
```

УКАЗАТЕЛ КЪМ ПОВЕЧЕ ОТ ЕДИН МЕТОДА

Когато добавяме методите Add и Multiply към делегата add с оператора +=, те се превръщат в множество от указатели към методи в делегата. При извикване на делегата add с параметри 3 и 5, ще бъдат извикани всички методи, съхранени в делегата, последователно. Резултатът от извикването на add ще бъде резултатът от последния метод във веригата - в случая Multiply.

```
1  delegate int Calculate(int x, int y);
2  ...
3  class Program
4  {
5      static void Main(string[] args)
6      {
7          Calculate add = Add;
8          add += Multiply;
9
10         int result1 = add(3, 5);
11
12         Console.WriteLine("Add result: " + result1); // 15
13     }
14
15     static int Add(int x, int y)
16     {
17         return x + y;
18     }
19
20     static int Multiply(int x, int y)
21     {
22         return x * y;
23     }
24 }
```

КАК ДА ИНСТАНЦИРАМЕ



```
1  delegate int Calculate(int x, int y);
2
3  static int Add(int x, int y)
4  {
5      return x + y;
6  }
7
8  Calculate add = Add;
```

```
1  delegate int Calculate(int x, int y);
2
3  Calculate add = delegate(int x, int y)
4  {
5      return x + y;
6  };
```

```
1  delegate int Calculate(int x, int y);
2
3  Calculate add = (x, y) => x + y;
```

EVENTS

Event-ите са механизми за уведомяване на обектите, когато се случи определено събитие. В C# event-ите са дефинирани като членове на класове и използват ключовата дума "event". Те могат да имат множество обработчици (handlers), които се изпълняват, когато събитието се вдигне.

```
// Дефиниране на event аргументи за събитието "ColorChanged"
public class ColorChangedEventArgs : EventArgs
{
    public string Color { get; set; }
}

public class TrafficLight
{
    // Дефиниране на event за събитието "ColorChanged"
    public event EventHandler<ColorChangedEventArgs> ColorChanged;

    public void ChangeColor(string newColor)
    {
        Console.WriteLine($"Цветът на светофара е променен на {newColor}.");
        ColorChanged?.Invoke(this, new ColorChangedEventArgs { Color = newColor });
    }
}

public class Car
{
    public void OnColorChanged(object sender, ColorChangedEventArgs e)
    { Console.WriteLine($"Светофарът е {e.Color}. Колата трябва да {GetInstruction(e.Color)}."); }

    private string GetInstruction(string color)
    {
        switch (color)
        {
            case "червен":
                return "спре";
            case "жълт":
                return "намали скоростта";
            default:
                return "не прави нищо";
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        TrafficLight trafficLight = new TrafficLight();
        Car car1 = new Car();
        // Прикачане на метода който да се изпълнява при събитието "ColorChanged"
        trafficLight.ColorChanged += car1.OnColorChanged;
        trafficLight.ChangeColor("червен");
```

LOGGERS

Loggers са инструменти, които позволяват на програмистите да записват и съхраняват информация за различни дейности и събития, които се извършват в програмния код. Това могат да бъдат грешки, изключения, действия на потребителите и други.

Loggers са особено полезни при отстраняването на грешки в приложенията и при подобряване на техния функционал. Чрез записване на информацията за грешките, програмистите могат да открият и да разрешат проблемите в приложението си, както и да подобрят производителността на приложението.

```
using Microsoft.Extensions.Logging;
using System;

class ConsoleLogger : ILogger
{
    public IDisposable BeginScope<TState>(TState state) => null

    public bool IsEnabled(LogLevel logLevel) => true

    public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Ex
    {
        Console.WriteLine(DateTime.Now.ToString() + " - " + formatter(state, exc
    }
}

class Program
{
    static void Main(string[] args)
    {
        ILogger logger = new ConsoleLogger();

        logger.LogInformation("Program started.");
        try
        {
            //...
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error occurred.");
        }

        logger.LogInformation("Program ended.");
    }
}
```

INTERFACES

Интерфейсите в C# са абстрактни дефиниции на методи и свойства, които класовете и други типове могат да имплементират. Те представляват контракт между две части на програмата, като определят какви методи и функционалности е нужно да имплементират обектите, които ги използват. Използването на интерфейси позволява да се постигне полиморфизъм и абстракция, като позволява работа с обекти на различни класове чрез общ интерфейс.

```
interface IParty
{
    void Dance();
    void Drink();
}

class HouseParty : IParty
{
    public void Dance() => Console.WriteLine("Dance!");

    public void Drink() => Console.WriteLine("Drink!");
}

class ClubParty : IParty
{
    public void Dance() => Console.WriteLine("Dance!");

    public void Drink() => Console.WriteLine("Drink & More!");
}

class Program
{
    static void Main(string[] args)
    {
        IParty[] houseParty = new IParty[2];
        houseParty[0] = new HouseParty();
        houseParty[1] = new ClubParty();

        foreach (var party in houseParty)
        {
            party.Dance();
            party.Drink();
        }
```

CONCURRENTDICTIONARY

ConcurrentDictionary е клас в C#, който предоставя безопасен начин за манипулация на речник (dictionary) в многонишкова среда. Той е част от пространството за имена System.Collections.Concurrent и позволява паралелна работа с речници, като осигурява безопасност на нишковете, които имат достъп до данните. Това го прави подходящ за приложения, които имат нужда от ефективно и безопасно паралелно четене и записване на данни в речник.

```
ing System;
ing System.Collections.Concurrent;
ing System.Threading.Tasks;

lass Program
{
    static void Main(string[] args)
    {
        ConcurrentDictionary<int, string> dictionary = new ConcurrentDictionary<in

        Task task1 = Task.Factory.StartNew(() =>
        {
            for (int i = 0; i < 1000; i++)
            {
                dictionary.TryAdd(i, "value " + i.ToString());
            }
        });

        Task task2 = Task.Factory.StartNew(() =>
        {
            for (int i = 1000; i < 2000; i++)
            {
                dictionary.TryAdd(i, "value " + i.ToString());
            }
        });

        Task.WaitAll(task1, task2);

        foreach (KeyValuePair<int, string> pair in dictionary)
        {
            Console.WriteLine(pair.Key + ": " + pair.Value);
        }
    }
```

LISTS

В C# List е клас, който представлява динамичен масив от обекти, който може да се увеличава или намалява в зависимост от нуждите на приложението. List-овете са също така гъвкави, тъй като могат да съдържат обекти от различни типове. Може да се достъпват елементите на List по индекс, като е възможно да се вмъкват, изтриват и търсят елементи.

ВИДОВЕ ЛИСТОВЕ

1. List<T> - обичайният динамичен масив от обекти от тип T
2. LinkedList<T> - двусвързан списък от обекти от тип T
3. SortedList<TKey, TValue> - речник, който е сортиран според ключа TKey и може да съдържа стойности от тип TValue
4. Stack<T> - колекция, която позволява само добавяне на елементи и премахване на последния добавен елемент (по принцип "последен влезе, първи излезе" или "Last-In-First-Out")
5. Queue<T> - колекция, която позволява само добавяне на елементи и премахване на първия добавен елемент (по принцип "първи влезе, първи излезе" или "First-In-First-Out")

Всички тези листове предоставят различни функционалности, които могат да бъдат полезни в зависимост от конкретните нужди на приложението.

COLLECTIONS

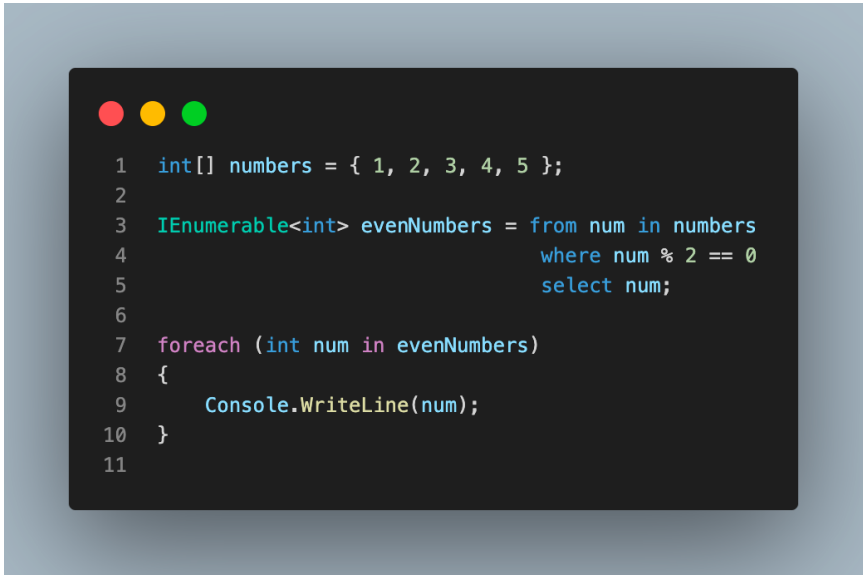
Колекциите в C# са групи от обекти, които могат да бъдат манипулирани като едно цяло. Те предоставят по-голяма функционалност и удобство при работа с множества от обекти, като осигуряват определени операции като добавяне, премахване, търсене и сортиране на елементите.

ПРИМЕРНИ КОЛЕКЦИИ

1. List<T> - динамичен масив от обекти от тип T
2. LinkedList<T> - двусвързан списък от обекти от тип T
3. Queue<T> - колекция, която позволява само добавяне на елементи и премахване на първия добавен елемент (по принцип "първи влезе, първи излезе" или "First-In-First-Out")
4. Stack<T> - колекция, която позволява само добавяне на елементи и премахване на последния добавен елемент (по принцип "последен влезе, първи излезе" или "Last-In-First-Out")
5. Dictionary<TKey, TValue> - речник, който асоциира ключове от тип TKey със стойности от тип TValue
6. HashSet<T> - множество от уникални обекти от тип T
7. SortedList<TKey, TValue> - речник, който е сортиран според ключа TKey и може да съдържа стойности от тип TValue
8. ObservableCollection<T> - колекция от обекти от тип T, която автоматично известява всички подписани на нея клиенти при промяна на съдържанието ѝ

LINQ

Linq (Language Integrated Query) е технология, която позволява на програмистите да използват SQL подобни заявки за търсене и манипулиране на данни, като се използва статичен тип безопасен синтаксис в C#. Linq поддържа много типове данни и източници на данни, включително колекции, масиви, бази данни, XML документи и други. Това прави Linq много полезен инструмент за програмистите, който може да помогне при работата с данни във всички видове приложения.



```
1  int[] numbers = { 1, 2, 3, 4, 5 };
2
3  IEnumerable<int> evenNumbers = from num in numbers
4                                 where num % 2 == 0
5                                 select num;
6
7  foreach (int num in evenNumbers)
8  {
9      Console.WriteLine(num);
10 }
11
```

LAMBDA

Lambda изразите в C# са анонимни функции, които могат да бъдат присвоени на променлива и използвани за създаване на делегати или за използване в Linq заявки. Те се използват, когато е нужно да се предаде функционалност като параметър на друг метод или когато е необходимо да се създаде нова функционалност на място. Lambda изразите в C# са удобни, тъй като намаляват необходимостта от допълнително деклариране на методи и улесняват четимостта на кода.

```
1  List<string> names = new List<string> { "Alice", "Bob", "Charlie", "Dave" };
2
3  List<string> shortNames = names.FindAll(x => x.Length < 5);
4
5  foreach (string name in shortNames)
6  {
7      Console.WriteLine(name);
8  }
9
```

EXTENSION METHODS

Разширителните методи (extension methods) в C# са методи, които могат да бъдат прикачени към съществуващ клас или интерфейс, без да се променя оригиналния код на този клас или интерфейс. Това позволява на програмистите да добавят функционалност към съществуващ код, без да го променят или да създават нов клас.

Разширителните методи се декларират като статични методи в статичен клас, който може да бъде наречен като "клас на разширителните методи". Те могат да се използват като методи на оригиналния клас, като се предостави пълното име на метода включващо името на класа или интерфейса, към който е прикачен разширителният метод.

ПРИМЕР

```
1   public class Person
2   {
3       public string Name { get; set; }
4       public DateTime Birthdate { get; set; }
5
6       public Person(string name, DateTime birthdate)
7       {
8           Name = name;
9           Birthdate = birthdate;
10      }
11  }
12
13  public static class PersonExtensions
14  {
15      public static int GetAge(this Person person)
16      {
17          DateTime now = DateTime.Today;
18          int age = now.Year - person.Birthdate.Year;
19          if (person.Birthdate > now.AddYears(-age)) age--;
20          return age;
21      }
22  }

Person person = new Person("John", new DateTime(1980, 1, 1));
int age = person.GetAge();
```

АТРИБУТИ

Атрибутите в C# са специални маркери, които се прилагат върху кода, за да му се добави метаданни или за да му се дадат инструкции за обработка или генериране на код.

Атрибутите в C# се декларират като класове, които са маркирани с [ ] скоби. Те могат да бъдат прикачени към различни елементи на кода, като класове, методи, свойства, параметри и т.н.

```
using System;

class Program
{
    [Obsolete("Use newMethod instead", false)]
    static void OldMethod()
    {
        Console.WriteLine("This method is obsolete.");
    }

    static void NewMethod()
    {
        Console.WriteLine("This is the new method.");
    }

    static void Main(string[] args)
    {
        OldMethod(); // warning message will be shown at compile tim
        NewMethod();
    }
}
```

ДЕКЛАРИРАНЕ НА АТРИБУТ

```
1  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
2  public class MyCustomAttribute : Attribute
3  {
4      private string myCustomValue;
5
6      public MyCustomAttribute(string myCustomValue)
7      {
8          this.myCustomValue = myCustomValue;
9      }
10
11     public string MyCustomValue
12     {
13         get { return myCustomValue; }
14     }
15 }
```

Можете да дефинирате собствени атрибути в C# като декларирате нов клас, който е наследник на System.Attribute класа.

USING

Ключовата дума using в C# се използва за автоматично управление на ресурси, които трябва да бъдат освободени след използването им, като например файлове, бази данни, мрежови връзки и други. Когато използваме using за даден ресурс, той автоматично се освобождава след като приключи работата му, дори и да има грешка или изключение по време на изпълнението на кода. Това прави кода по-сигурен и по-ефективен, тъй като гарантира, че ресурсите се освобождават навреме и не остават заети за дълго време.

```
using (var context = new DatabaseContext())
{
    context.Database.EnsureCreated();
    context.Add<DatabaseUser>(new DatabaseUser()
    {
        Name = "user",
        Password = "password",
        Expires = DateTime.Now,
        Role = UserRolesEnum.Student
    });
    context.SaveChanges();
    var users = context.Users.ToList();
    Console.ReadKey();
}
```

WPF

WPF е съкращение от Windows Presentation Foundation и е технология за разработка на десктоп приложения за операционните системи на Windows. WPF използва XAML (Extensible Application Markup Language) за декларативно описване на графичния интерфейс на потребителя, а в кода се използва C# или друг език от .NET платформата за програмиране на поведението на приложението. WPF осигурява гъвкав и мощен подход за създаване на мултимедийни, графични и интерактивни потребителски интерфейси с поддръжка на векторна графика, анимации, стилове и теми.

BINDING

Binding в WPF е механизъм за свързване на данни между два обекта, като единият от тях представлява източник на данни, а другият е получател на тези данни. Binding се използва за автоматично обновяване на UI елементите (например текстови кутии, лейбъли, списъци и др.), когато източникът на данни се промени.

Binding в WPF се осъществява посредством класове като Binding, BindingExpression, BindingOperations и др. Обикновено се използва в XAML кода, като се указва източникът на данни, свойството, което да се свърже и вида на Binding (например OneWay, TwoWay, OneTime и др.).

Когато Binding е установен, WPF генерира BindingExpression, който се грижи за поддръжката на връзката между източника на данни и получателя на тези данни. BindingExpression следи промените в източника на данни и автоматично ги отразява в UI елемента. Ако се направят промени в UI елемента, BindingExpression ги връща към източника на данни, ако е зададен вида на Binding, който позволява това.

Binding в WPF е мощен механизъм, който улеснява работата с данни в потребителския интерфейс и позволява лесно и автоматично обновяване на данните в реално време.