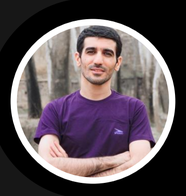
Key Features review

5

6

7

8



Nabi Karampoor
@thisisnabi

1 String Interpolation

```
string name = "John";
int age = 30;
string result = name + " is " + age + " years old.";
```

String Concatenation

String Format

```
string name = "John";
int age = 30;
string result = string.Format("{0} is {1} years old.", name, age);
```

```
string name = "John";
int age = 30;
string result = $"{name} is {age} years old.";
```

String Interpolation

It allows you to create more readable and concise string formatting.

Nabi Karampoor
@thisisnabi

2 Null Conditional Operator

You would need to explicitly check each level for null before accessing nested members.

```
string result = (person != null && person.Address != null) ?
                person.Address.City :
                null;
```

?. Operator

Improving code readability and reducing the likelihood of null reference errors.

```
string result = person?.Address?.City;
```

Nabi Karampoor
@thisisnabi

3 Null-coalescing

```
int? nullableValue = GetNullableValue();
int result;

if(nullableValue != null)
{
      result = nullableValue;
}
else
{
      result = 10;
}
```

Before that you must typically involves using conditional checks and providing default values.

```
int? nullableValue = GetNullableValue();
int result = 10;

if (nullableValue != null)
{
    result = nullableValue.Value;
}
```

```
int? nullableValue = GetNullableValue();
int result = nullableValue ?? 10;
```

?? Operator

Provide a default value for a nullable expression.

Nabi Karampoor
@thisisnabi

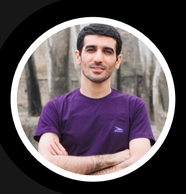
4 Auto-Implemented Properties

```
public class Person
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

you manually declare private backing fields and write explicit getter and setter methods for each property.

```
public class Person
{
    public string name {get; set;}

    // initialize auto-implemented
    // public string name {get; set;} = "Diman";
}
```

Shorthand syntax for defining properties without explicitly declaring the backing field.



Nabi Karampoor
@thisisnabi

5 Expression-bodied

```
public class Person
{
    private int age;
    public int Age
    {
        get { return age; }
        set { age = value >= 0 ? value : 0; }
    }

    public void IncrementAge()
    {
        Age++;
    }
}
```

Separate block is used for the method body or separate getter and setter blocks are used

```
public class Person
{
    private int age;
    public int Age
    {
        get => age;
        set => age = value >= 0 ? value : 0;
    }

    public void IncrementAge() => Age++;
}
```

Provide a concise syntax for defining methods and properties.

Nabi Karampoor
@thisisnabi

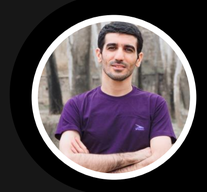
6 Collection Initializer

```
List<Person> people = new List<Person>();
people.Add(new Person { Name = "Alice", Age = 25 });
people.Add(new Person { Name = "Bob", Age = 30 });
people.Add(new Person { Name = "Charlie", Age = 22 });
```

you would create a collection and add elements to it individually.

```
List<Person> people = new List<Person>
{
      new Person { Name = "Alice", Age = 25 },
      new Person { Name = "Bob", Age = 30 },
      new Person { Name = "Charlie", Age = 22 }
};
```

Provides a concise way to initialize collections, such as arrays, lists, and dictionaries.



Nabi Karampoor
@thisisnabi

7 Static Using Statement

```
public static class MathOperations
{
    public static int Add(int a, int b) => a + b;
    public static int Subtract(int a, int b) => a - b;
}

class Program
{
    using static MathOperations;

    static void Main()
    {
        // Using static members without specifying the type name
        int sum = Add(5, 3);
        int difference = Subtract(8, 4);

        Console.WriteLine($"Sum: {sum}, Difference: {difference}");
    }
}
```

Allows you to access static members of a type without specifying the type name each time.

Nabi Karampoor
@thisisnabi