Key Features review

C#

8

9

10

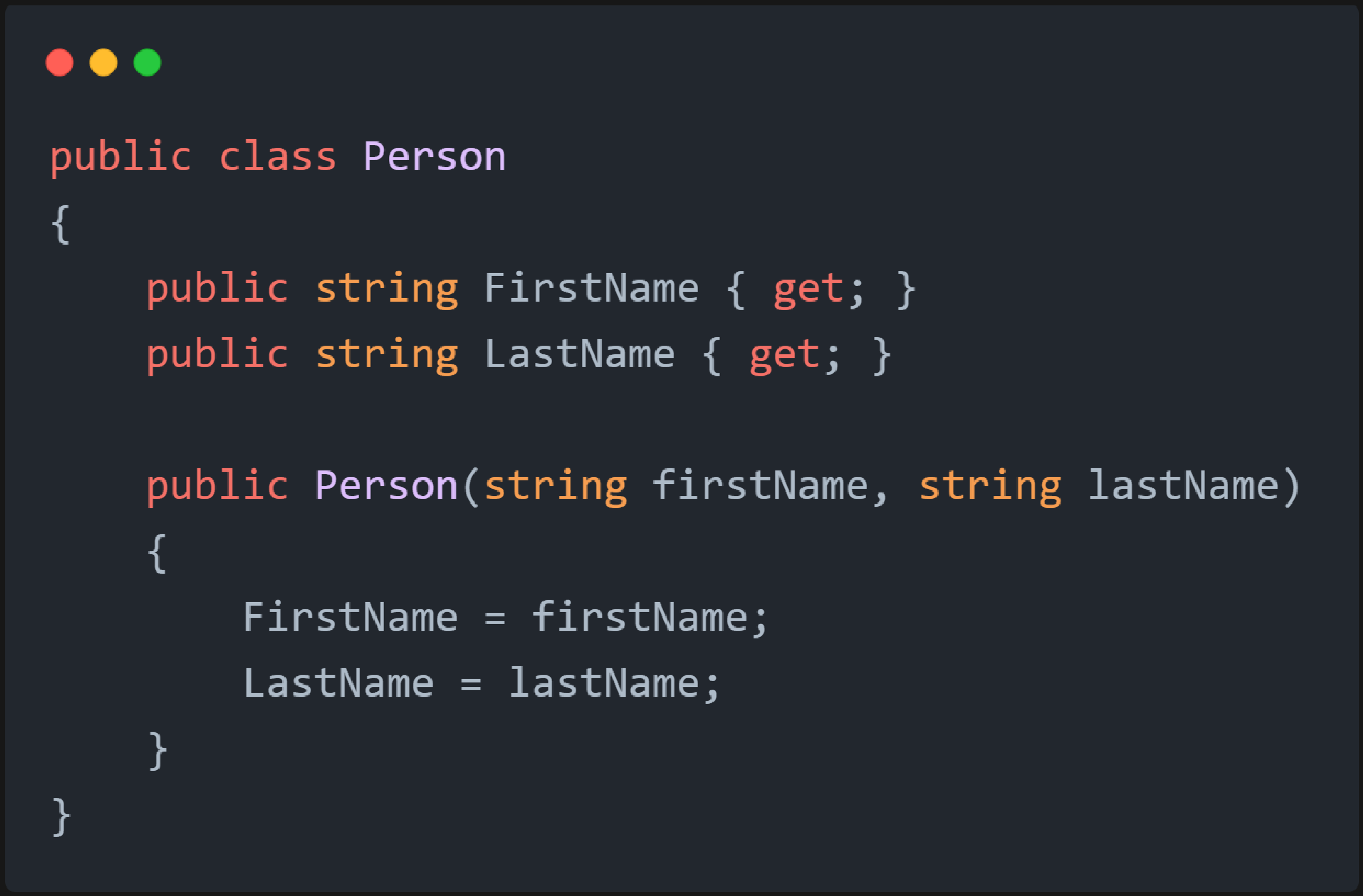
11

Nabi Karampoor
@thisisnabi

1 1

Records

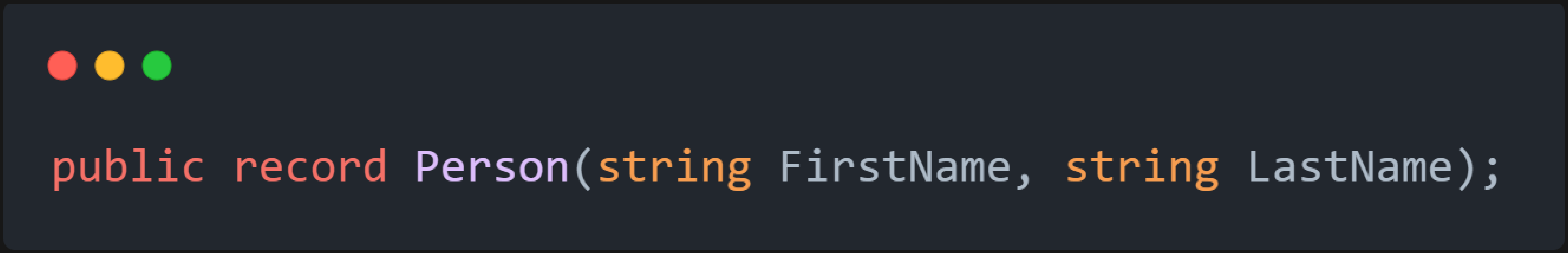
Immutability



```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

Unchangeable State

Concise way to define immutable types



```
public record Person(string FirstName, string LastName);
```

Nabi Karampoor
@thisisnabi

1
2

Records

Value equality

Structural equality

The equality of their structure or content rather than their identity or reference.

```
public class Person {
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) {
        FirstName = firstName;
        LastName = lastName;
    }

    public override bool Equals(object obj) {
        if (obj is not Person otherPerson)
            return false;

        return FirstName == otherPerson.FirstName && LastName == otherPerson.LastName;
    }

    public override int GetHashCode() {
        return HashCode.Combine(FirstName, LastName);
    }
}
```

```
public record Person(string FirstName, string LastName);
```

auto-generated value-based equality.

```
var person1 = new Person ("John","Doe");
var person2 = new Person ("John","Doe");

// Equality check using generated equality implementation
bool areEqual = person1 == person2; // true
```

Nabi Karampoor
@thisisnabi

2 init-only Setter

```csharp
public class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

Valid during initialization

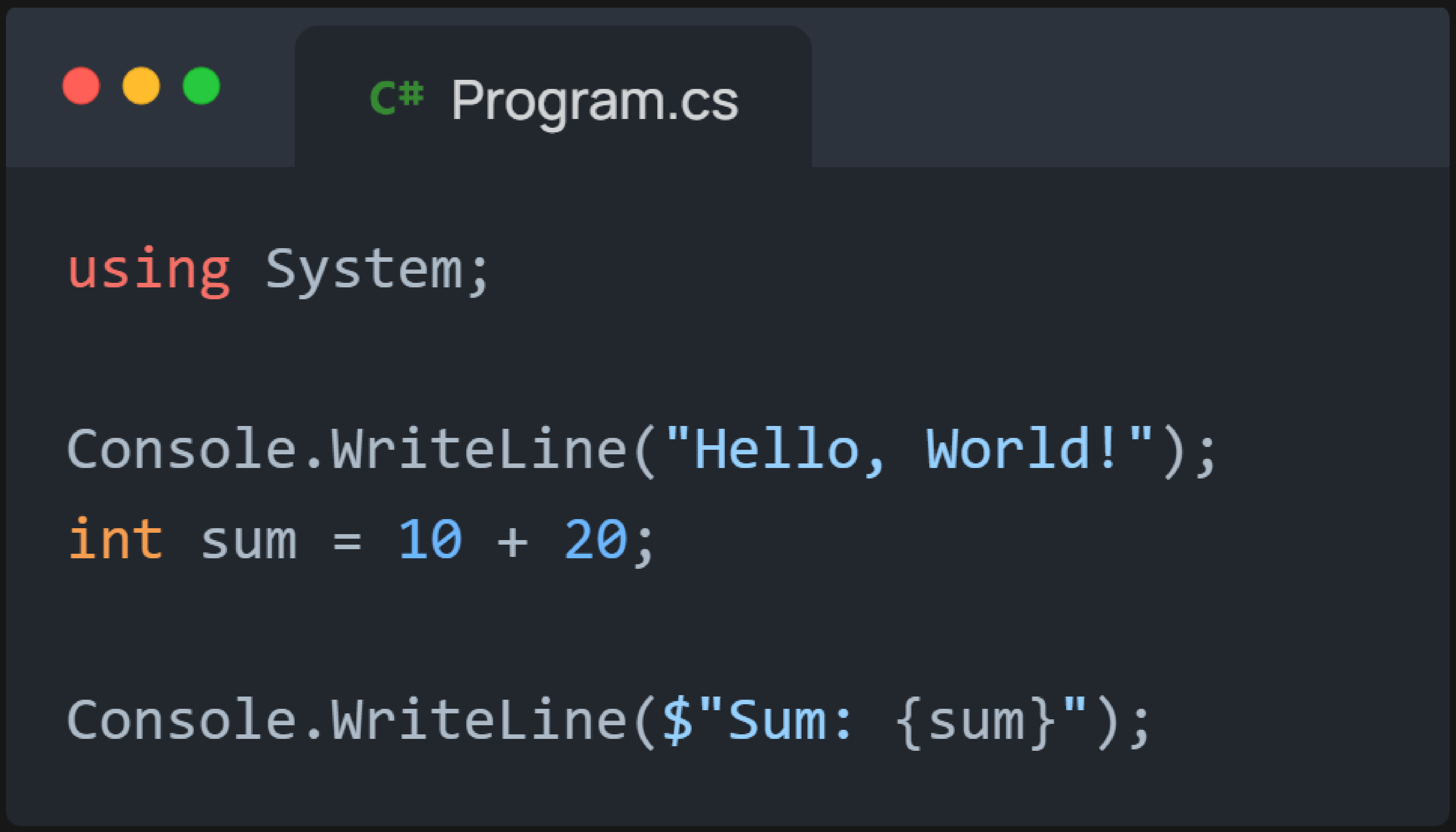
```csharp
var person = new Person
{
    FirstName = "John",
    LastName = "Doe"
};

Console.WriteLine($"Full Name: {person.FirstName} {person.LastName}");

// person.FirstName = "Jane";
```

Cannot modify init-only property

Nabi Karampoor
@thisisnabi

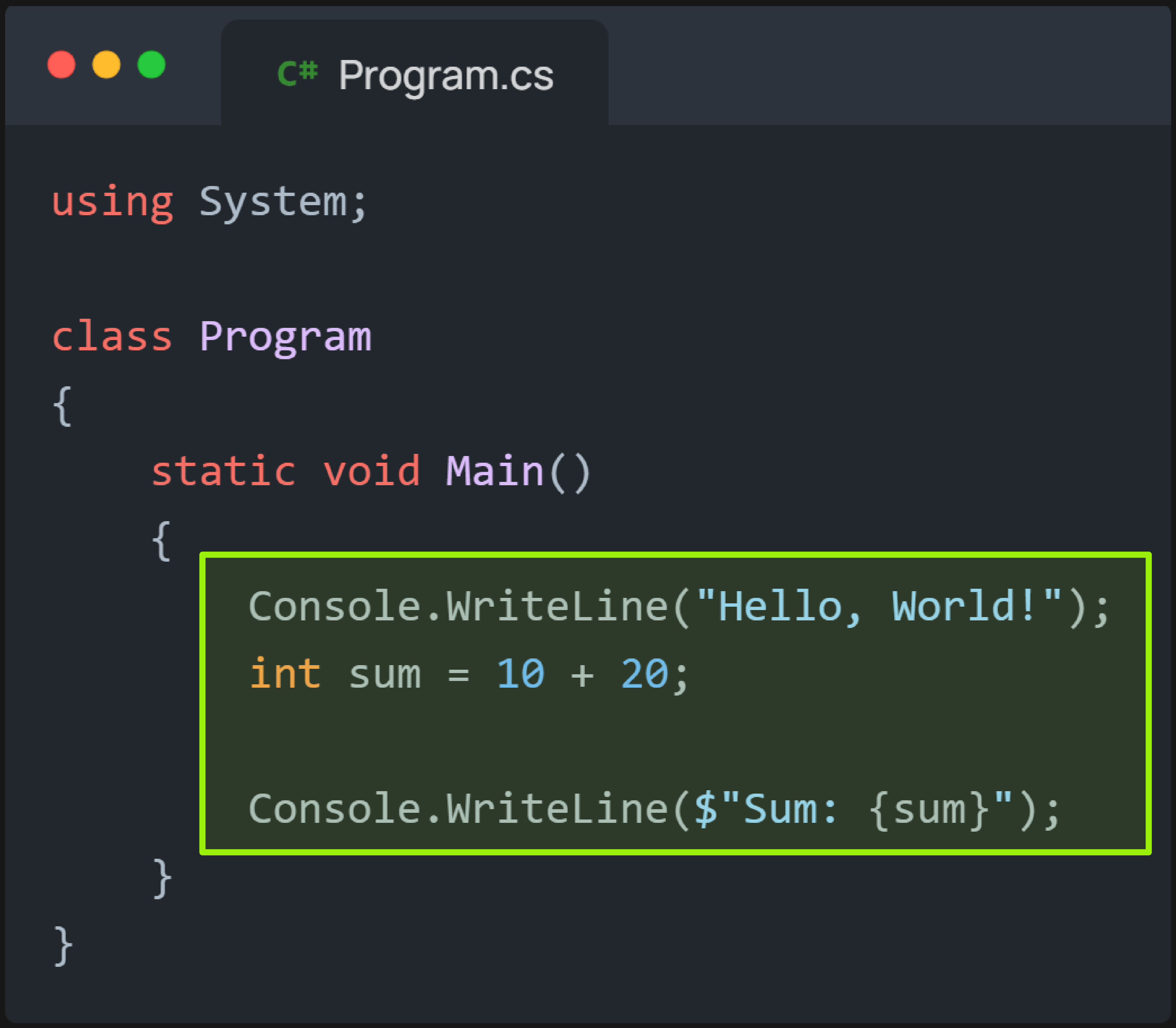
3 Top Level Statement

Great addition for simplifying the entry point of small console applications, but for larger projects, the traditional approach with a class and Main method offers a more structured.



Program.cs

```csharp
using System;

Console.WriteLine("Hello, World!");
int sum = 10 + 20;

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



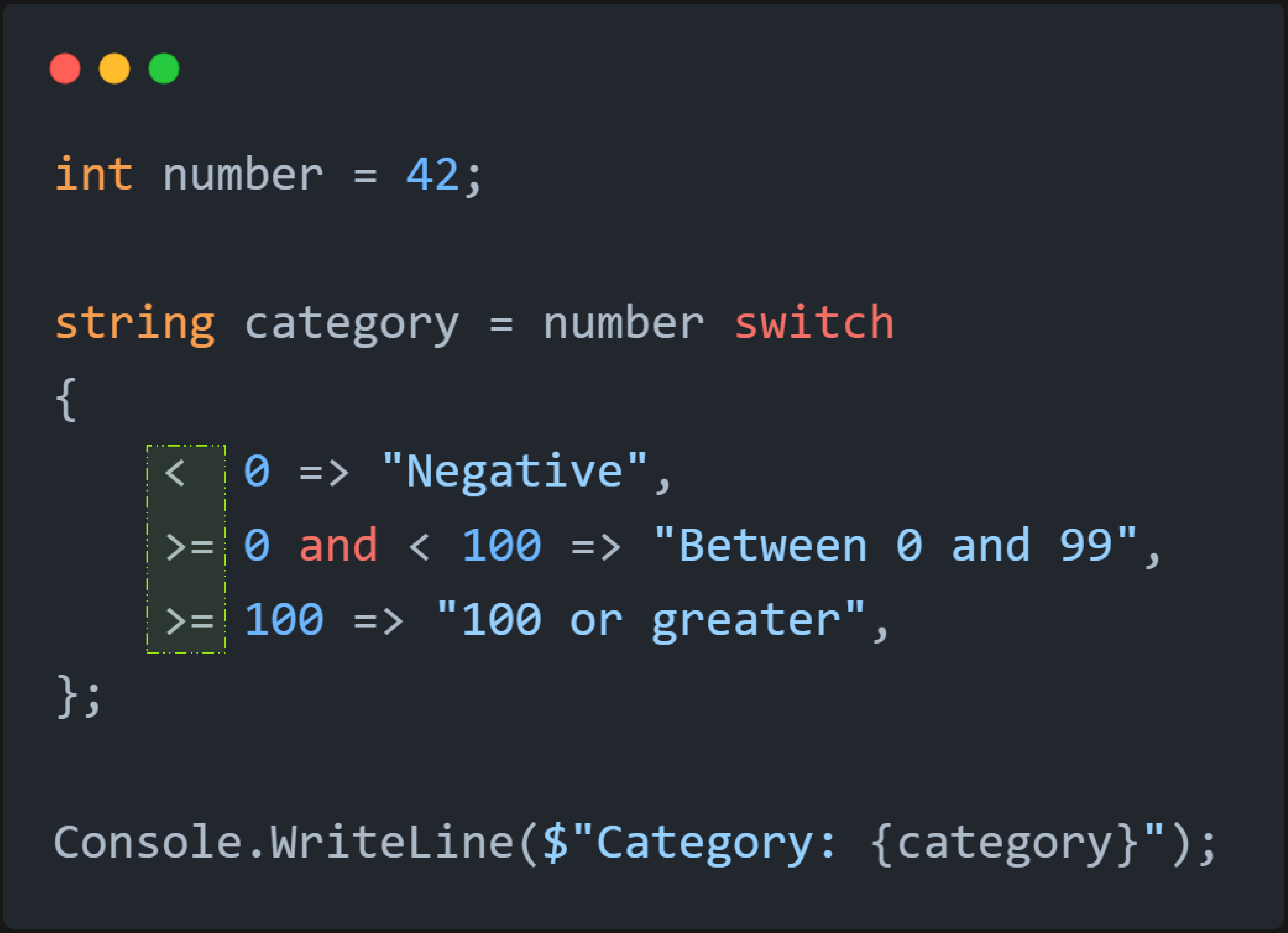
Program.cs

```csharp
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, World!");
        int sum = 10 + 20;

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

Nabi Karampoor
@thisisnabi

Relational patterns

Enhanced Pattern Matching



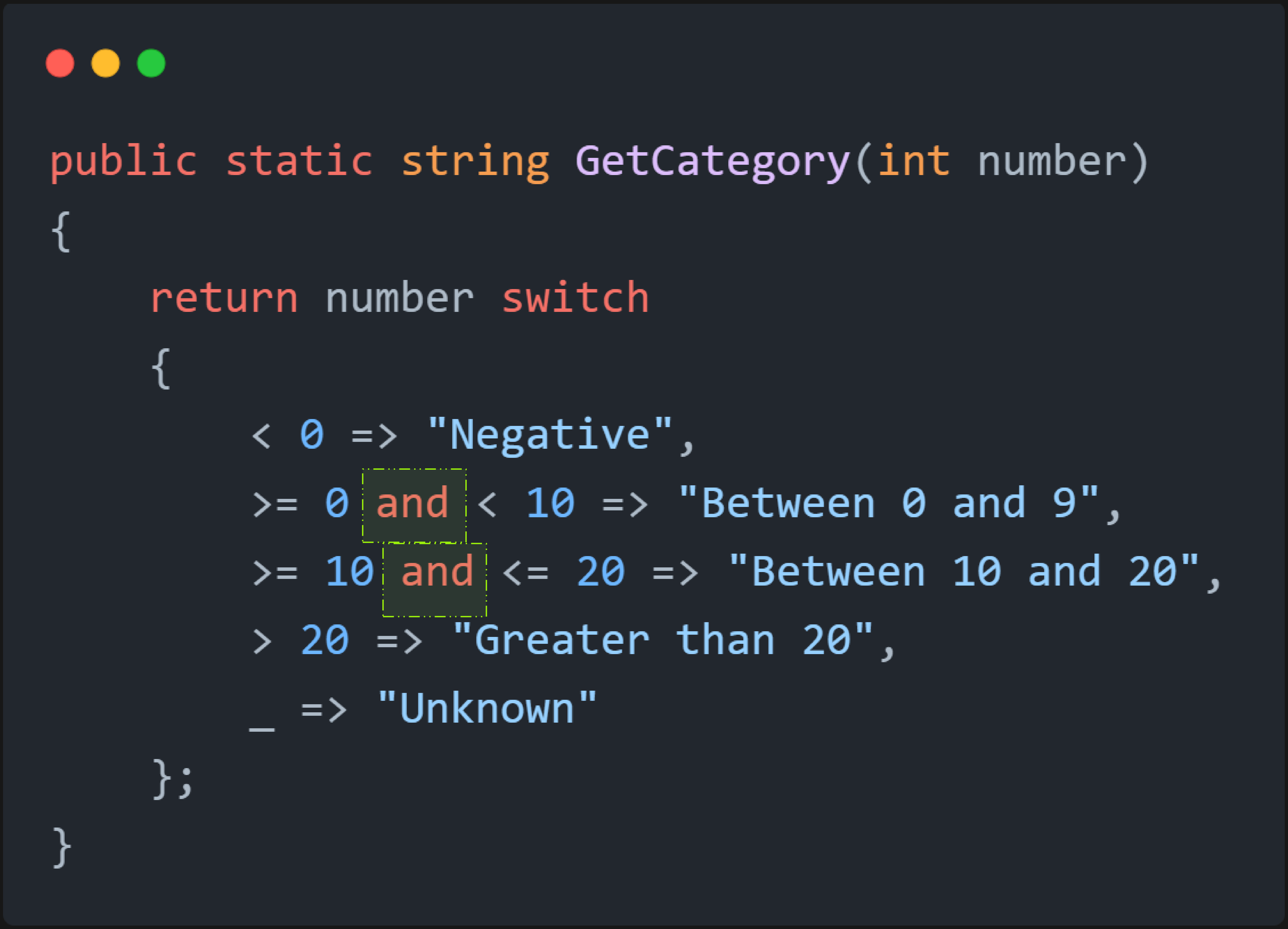
```
int number = 42;

string category = number switch
{
    <  0 => "Negative",
    >= 0 and < 100 => "Between 0 and 99",
    >= 100 => "100 or greater",
};

Console.WriteLine($"Category: {category}");
```

Useful when you want to check if a value falls within a specific range.

Nabi Karampoor
@thisisnabi

Logical patterns

Enhanced Pattern Matching



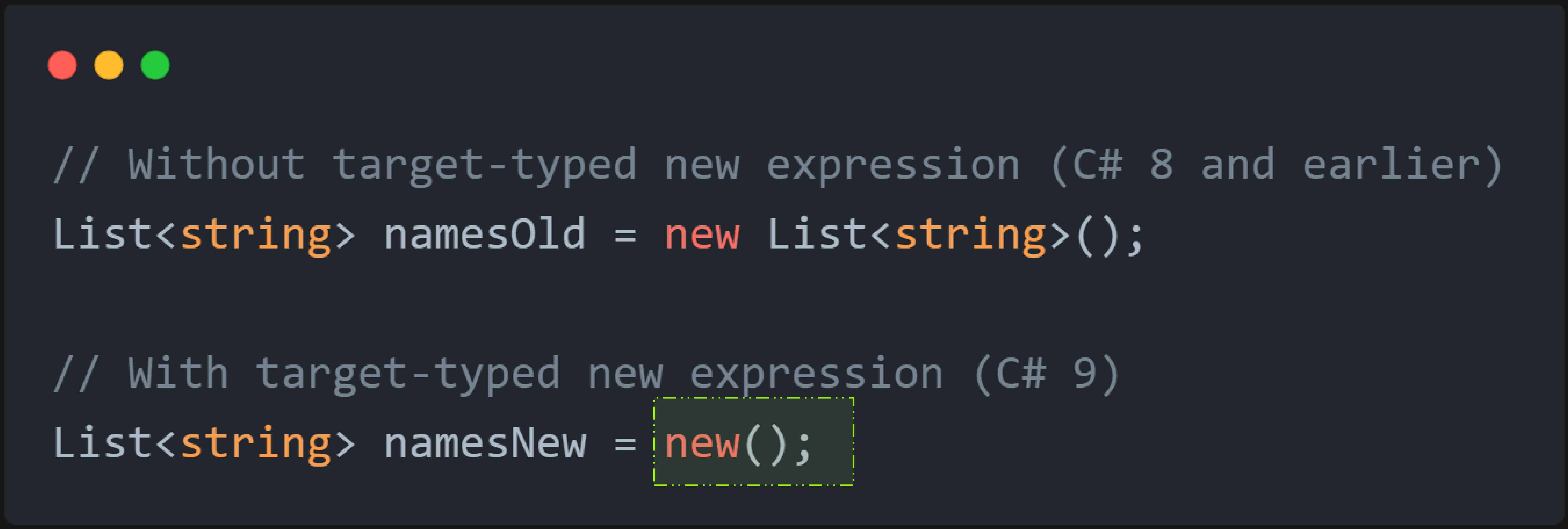
```
public static string GetCategory(int number)
{
    return number switch
    {
        < 0 => "Negative",
        >= 0 and < 10 => "Between 0 and 9",
        >= 10 and <= 20 => "Between 10 and 20",
        > 20 => "Greater than 20",
        _ => "Unknown"
    };
}
```

combine patterns using logical operators like and, or, not.

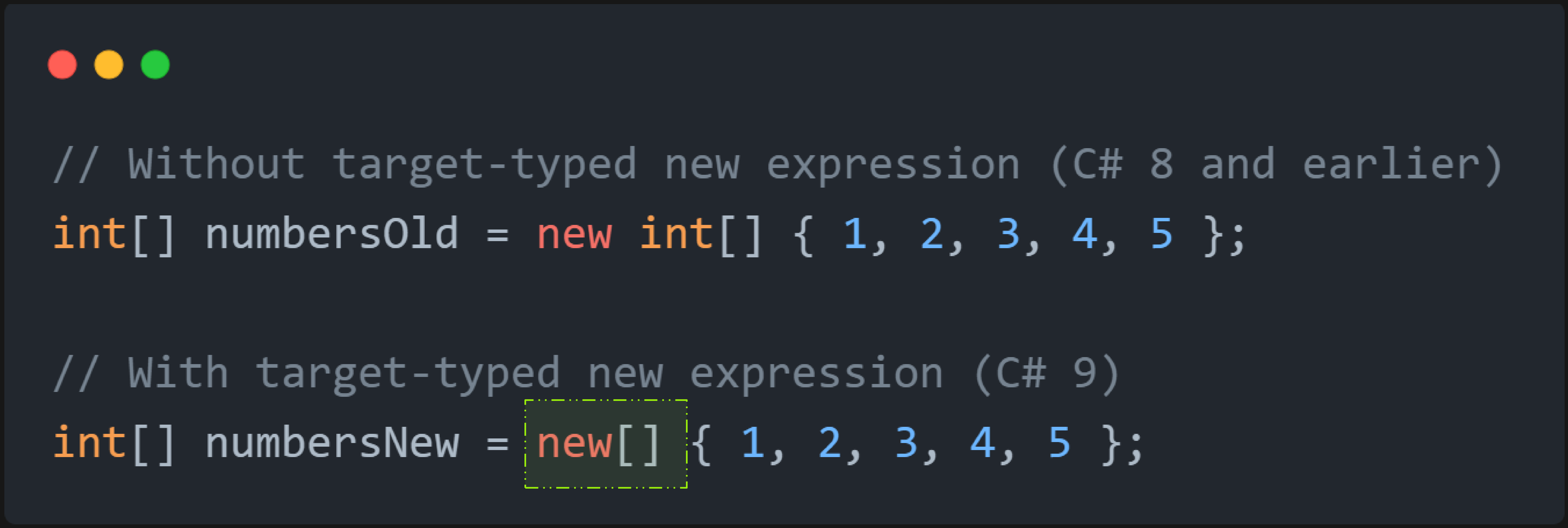
Nabi Karampoor
@thisisnabi

5 Target-typed new

Simplify the syntax when creating instances of objects by allowing the type to be inferred from the context



```csharp
// Without target-typed new expression (C# 8 and earlier)
List<string> namesOld = new List<string>();

// With target-typed new expression (C# 9)
List<string> namesNew = new();
```



```csharp
// Without target-typed new expression (C# 8 and earlier)
int[] numbersOld = new int[] { 1, 2, 3, 4, 5 };

// With target-typed new expression (C# 9)
int[] numbersNew = new[] { 1, 2, 3, 4, 5 };
```

Nabi Karampoor
@thisisnabi