**New features introduced in C# 9:**

***Top-level programs:*** C# 9 introduces the ability to write top-level programs, which allow you to write C# code without the need for a class or method declaration. Here's an example:

*using System;*

*Console.WriteLine("Hello, world!");*

This program simply prints the string "Hello, world!" to the console. Because it's a top-level program, you don't need to define a class or method to encapsulate the code.

***Target-typed new expressions***: This allow you to omit the type name when creating a new instance of a class. Here's an example:

*var person = new Person("John", "Doe", 42);*

In this example, we're creating a new instance of the Person class, but we're able to omit the type name because the type can be inferred from the variable declaration.

***Init-only properties:*** C# 9 introduces init-only properties, which allow you to create properties that can only be set during object initialization. Here's an example:

*public class Person*

*{*

*public string FirstName { get; init; }*

*public string LastName { get; init; }*

*public int Age { get; init; }*

*}*

In this example, we're defining a Person class with three properties: FirstName, LastName, and Age. The init keyword in the property declaration indicates that the property can only be set during object initialization.

***Improved pattern matching:*** C# 9 introduces improved pattern matching for switch statements, which allow you to match on types, values, and patterns in a more concise and expressive way. Here's an example:

*switch (person)*

*{*

*case null:*

*Console.WriteLine("Person is null");*

*break;*

*case { FirstName: "John", LastName: "Doe" }:*

*Console.WriteLine("Person is John Doe");*

*break;*

*case { Age: var age }:*

*Console.WriteLine($"Person is {age} years old");*

*break;*

*default:*

*Console.WriteLine("Unknown person");*

*break;*

*}*

In this example, we're using a switch statement to match on a Person object. We're able to match on the properties of the object using pattern matching, and we're also able to extract the value of the Age property using a variable pattern.

**Records**: Records are a new type of reference type that were introduced in C# 9.0. They're designed to be more concise and expressive than traditional classes, and they're especially useful for modeling immutable data. Here's an example of how you might use a record to model a person:

*public record Person(string FirstName, string LastName, int Age);*

This defines a record called Person with three properties: FirstName, LastName, and Age. The properties are defined in the constructor, so they're automatically set when you create a new instance of the record.

***Target-typed conditional expressions***: the ability to use target-typed conditional expressions, which allow you to omit the type name when using the ternary operator. Here's an example:

var result = isEven ? "even" : "odd";

In this example, we're using the ternary operator to set the value of the result variable based on the value of the isEven variable. Because we've used a target-typed conditional expression, we're able to omit the type name of the result variable.

***Function pointers:*** C# 9 introduces function pointers, which allow you to define a variable that can hold a reference to a function. Here's an example:

*delegate int BinaryOperation(int x, int y);*

*int Add(int x, int y) => x + y;*

*int Subtract(int x, int y) => x - y;*

*BinaryOperation op = Add;*

*int result = op(3, 4); // result is 7*

*op = Subtract;*

*result = op(3, 4); // result is -1*

In this example, we're defining two functions (Add and Subtract) and a delegate (BinaryOperation) that can hold a reference to a function that takes two integers and returns an integer. We're then setting the value of the op variable to refer to each function in turn, and calling the op function pointer to compute the result.

***Lambda discard parameters***: C# 9 introduces lambda discard parameters, which allow you to discard parameters in lambda expressions using the \_ syntax. Here's an example:

*var numbers = new[] { 1, 2, 3, 4, 5 };*

*var evens = numbers.Where(n => n % 2 == 0);*

In this example, we're using a lambda expression to filter the numbers array to only include even numbers. We're using the \_ syntax to discard the first parameter of the lambda expression, since we don't need it in this case.

***Native sized integers:*** C# 9 introduces support for native sized integers, which allow you to use integer types that are the same size as the native integer type on the current platform. Here's an example:

*nint size = Environment.Is64BitProcess ? 8 : 4;*

*nint[] array = new nint[size];*

In this example, we're using the nint type to create an array of integers. The nint type is the same size as the native integer type on the current platform, so the size of the array will be either 4 or 8 bytes depending on whether the process is running in a 32-bit or 64-bit environment.

and and or logical patterns: C# 9 introduces and and or logical patterns, which allow you to combine patterns in a more expressive way. Here's an example:

*if (person is { FirstName: "John" } or { LastName: "Doe" })*

*{*

*Console.WriteLine("Person is John or Doe");*

*}*

In this example, we're using the or logical pattern to match on a Person object if either the FirstName property is "John" or the LastName property is "Doe".

Overall, these new features in C# 9 are designed to make the language more expressive and concise, while also improving the safety and reliability of your code. Whether you're using records to model immutable data or using pattern matching and function pointers to write more expressive code, these features can help you write more maintainable and scalable applications.

***New features from C#10:***

***Global using directives:*** In C# 10, you can now use global using directives to automatically import namespaces throughout your entire project. Here's an example:

global using System;

global using System.Collections.Generic;

List<int> numbers = new();

In this example, we use global using directives to import the System and System.Collections.Generic namespaces, allowing us to create a new List<int> object without explicitly specifying the namespace.

***File-scoped namespaces:*** With C# 10, you can define namespaces at the file level rather than the block level. This can help simplify your code and make it easier to manage. Here's an example:

namespace MyNamespace;

public class MyClass

{

// ...

}

In this example, we define the MyNamespace namespace at the file level rather than inside a block.

Interpolated strings with interpolated verbatim strings: C# 10 now allows you to use interpolated verbatim strings, which can be useful for including special characters or formatting in your strings. Here's an example:

string name = "John";

int age = 30;

string message = $@"My name is {name} and I am {age} years old.";

In this example, we use an interpolated verbatim string to include the values of the name and age variables in our message.

***Improved lambda expressions***: C# 10 introduces new features for lambda expressions, including support for tuple deconstruction and better inference of ref and out parameters. Here's an example:

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

numbers.ForEach((int i, out bool success) =>

{

success = i > 0;

Console.WriteLine($"Number: {i}, Success: {success}");

});

In this example, we use a lambda expression to iterate through the numbers list and print out each number along with a success flag indicating whether the number is greater than zero.

***Improved pattern matching:*** C# 10 improves pattern matching with the introduction of additional patterns and enhancements to existing patterns. The new patterns include:

*Logical patterns:* allows you to match patterns using logical operators such as 'and', 'or', and 'not'.

Relational patterns: allows you to match patterns based on relational operators such as '<', '<=', '>', and '>='.

Parenthesized patterns: allows you to group patterns together for more complex matching.

Here's an example that uses these new patterns:

if (value is (> 0 and < 10) or (>= 20 and <= 30))

{

Console.WriteLine("Value is in the range of 0-10 or 20-30");

}

In this example, we use logical and relational patterns to match values that are in the range of 0-10 or 20-30.

***Simplified field declaration syntax:*** C# 10 introduces a simplified field declaration syntax that allows you to declare fields with less boilerplate code. Here's an example:

*public class MyClass*

*{*

*public int MyProperty { get; init; } = 42;*

*}*

In this example, we use the new 'init' keyword to set the initial value of the MyProperty property to 42.

***Target-typed new expressions:*** C# 10 introduces target-typed new expressions, which allow you to omit the type name in new expressions when the type can be inferred from the context. Here's an example:

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

In this example, we use target-typed new expressions to create a new List<int> object without explicitly specifying the type name.

**Global using directives:** C# 10 introduces global using directives, which allow you to specify using directives that apply to all files in a project. This can be useful for reducing the amount of boilerplate code in your files. Here's an example:

*// global using directive*

*global using System;*

*class MyClass*

*{*

*void MyMethod()*

*{*

*Console.WriteLine("Hello, world!");*

*}*

*}*

In this example, we use a global using directive to import the System namespace into all files in the project. This allows us to use the Console class without specifying the namespace in our code.

***Lambda discard parameters:*** C# 10 allows you to use a discard parameter in lambda expressions. This allows you to write more concise code when you don't need to use a particular parameter in the lambda expression. Here's an example:

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

*int sum = numbers.Aggregate((\_, b) => \_ + b);*

In this example, we use a lambda discard parameter to indicate that we don't need to use the first parameter in the Aggregate method.

Extended property patterns: C# 10 extends property patterns to allow you to match on properties of nested objects. Here's an example:

*if (obj is { Name: "John", Address: { City: "New York" } })*

*{*

*Console.WriteLine("Object matches the pattern");*

*}*

In this example, we use a property pattern to match on the Name and Address properties of the obj object. We also use a nested property pattern to match on the City property of the Address property.

These are some of the new features introduced in C# 10. They aim to make the language more expressive, concise, and safer to use.

C# 11 features:

C# 11 is the latest version of the C# programming language, released in November 2021 as part of .NET 6. It includes several new features and improvements that aim to make the language more expressive, concise, and safe to use. In this document, I'll cover the most significant features of C# 11 and provide examples to demonstrate their usage.

File-scoped namespaces

In previous versions of C#, you had to define a namespace for every file in your project. This could lead to a lot of boilerplate code, especially in small projects. C# 11 introduces file-scoped namespaces, which allow you to define a namespace for a group of files. Here's an example:

// file1.cs

namespace MyNamespace;

partial class MyClass

{

void Method1() {}

}

// file2.cs

namespace MyNamespace;

partial class MyClass

{

void Method2() {}

}

In this example, we define a file-scoped namespace called MyNamespace that spans across two files, file1.cs and file2.cs. We also use the partial keyword to define a single class MyClass across the two files. This reduces the amount of boilerplate code and makes it easier to organize your codebase.

Improved pattern matching

Pattern matching is a powerful feature in C# that allows you to test if an object has a certain shape and extract data from it. C# 11 improves pattern matching with several new features.

a. Property patterns with assignment

In C# 11, you can use property patterns with assignment to deconstruct an object and assign its properties to variables. Here's an example:

if (person is { FirstName: "John", LastName: "Doe" } is var (firstName, lastName))

{

Console.WriteLine($"Hello, {firstName} {lastName}!");

}

In this example, we use a property pattern to match on the FirstName and LastName properties of a Person object. We also use the is var syntax to deconstruct the object and assign its properties to firstName and lastName variables.

b. Relational patterns

Relational patterns allow you to test if an object is equal to, greater than, or less than a value. Here's an example:

if (number is >= 0 and <= 100)

{

Console.WriteLine("Number is between 0 and 100");

}

In this example, we use a relational pattern to test if number is between 0 and 100.

c. Conjunctive and disjunctive patterns

Conjunctive and disjunctive patterns allow you to combine multiple patterns with logical operators. Here's an example:

if (person is ({ FirstName: "John" } or { LastName: "Doe" }) and not { Age: >= 18 })

{

Console.WriteLine("Person matches the pattern");

}

In this example, we use a conjunctive pattern to combine two property patterns with the or operator. We also use a negation pattern with the not operator to test if person is under 18 years old.

***Top-level programs***

Top-level programs allow you to write C# programs without a class or a Main method. This makes it easier to write simple programs, such as scripts and console applications. Here's an example:

using System;

Console.WriteLine("Hello, world!");

In this example, we use a top-level program to print "Hello,

Additionally, C# 11 introduces a new attribute, globalusing, which allows you to specify global aliases for commonly used namespaces across your entire project. This feature eliminates the need to add using statements to each file individually and improves the overall maintainability of your code.

For example, suppose you commonly use the System and System.Collections.Generic namespaces throughout your codebase. Instead of adding using statements to each file, you can define global aliases for these namespaces in your global.usings file:

global using System;

global using System.Collections.Generic;

Once you've defined these global aliases, you can use them in any file within your project without the need for explicit using statements:

// No explicit using statements needed

List<string> myList = new();

This can significantly reduce the amount of boilerplate code needed in your project and improve the readability of your code.

Overall, C# 11 introduces several new features that improve the overall productivity and maintainability of your codebase. The new interpolated strings, improved pattern matching, and global aliases can help you write more concise, readable, and maintainable code.

Another new feature in C# 11 is the **with** expression, which allows you to create a copy of an existing object with modified properties. This is particularly useful when working with immutable objects, as it allows you to easily create new instances with updated values without modifying the original object.

Here's an example of how to use the with expression:

*public record Person(string FirstName, string LastName, int Age);*

*// Create a new Person instance*

*var person = new Person("John", "Doe", 30);*

*// Create a new instance with an updated Age value*

*var updatedPerson = person with { Age = 31 };*

*// The original person object is not modified*

*Console.WriteLine(person.Age); // 30*

*// The updatedPerson object has the new Age value*

*Console.WriteLine(updatedPerson.Age); // 31*

In this example, we define a new Person record that contains three properties: FirstName, LastName, and Age. We then create a new instance of Person with the new keyword and assign it to the person variable.

Next, we use the with expression to create a new instance of Person with an updated Age value. The syntax { Age = 31 } creates a new anonymous object with a single property Age and a value of 31. The with keyword then creates a new instance of Person with the same values as the original person object, except with the Age property updated to 31.

The original person object is not modified by this operation, and we can verify that its Age property still has the original value of 30. The updatedPerson object, on the other hand, has the new Age value of 31.

The with expression can be particularly useful when working with complex objects that contain many properties, as it allows you to easily create modified copies of those objects without having to write boilerplate code to copy each property individually.