

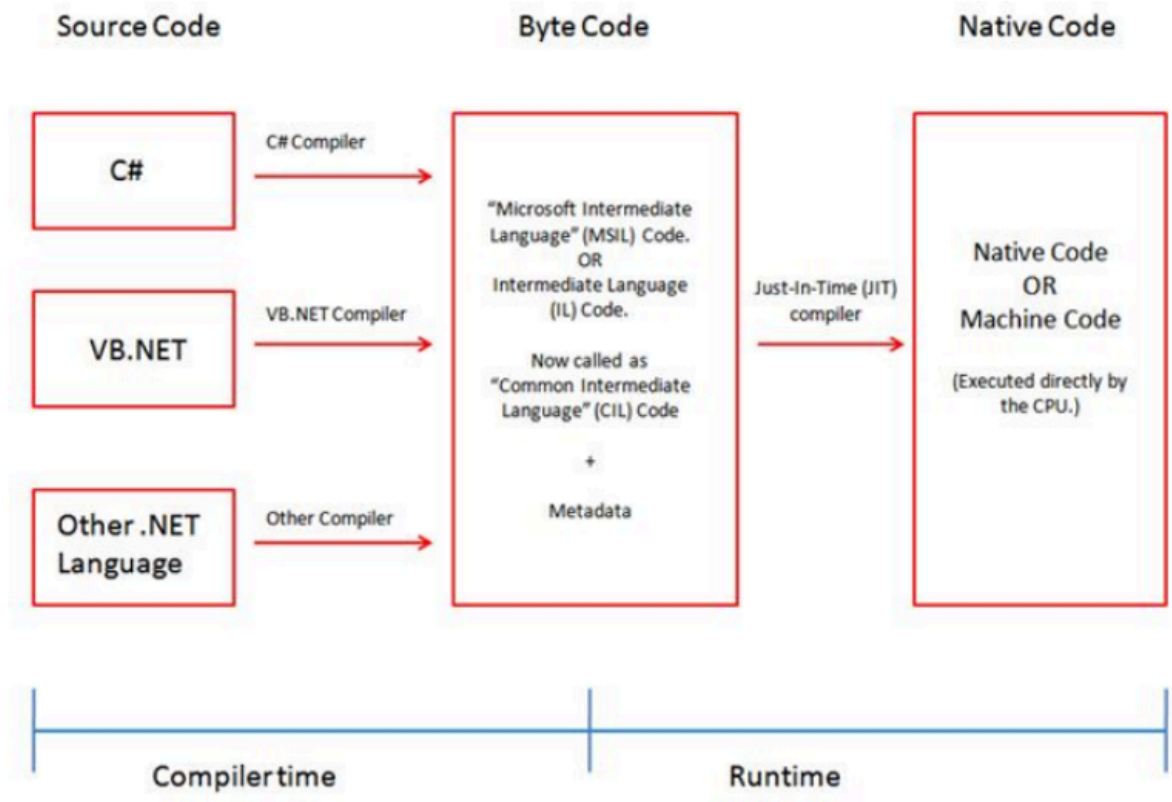
1. .NET platform

The .NET platform, created by Microsoft, is a comprehensive framework designed for developing and running applications on Windows. It was introduced in the late 1990s, with the first version released in 2002. The creation of .NET was driven by several key objectives and considerations:

- 1. Language Interoperability:** One of the primary reasons for creating .NET was to facilitate language interoperability. Before .NET, software developers often faced challenges when trying to integrate systems written in different programming languages. .NET introduced a Common Language Runtime (CLR) that allows applications written in different programming languages to work together seamlessly. This means developers can use the language best suited to their application's needs while still ensuring compatibility.
- 2. Simplified Development:** .NET aimed to simplify the development process for Windows applications. By providing a unified class library, developers can access a vast range of pre-written code, significantly reducing the amount of code they need to write from scratch. This library offers consistent access to database connectivity, web application development, desktop application development, and more, streamlining the development process.
- 3. Modern Application Requirements:** The platform was designed with modern application requirements in mind, including web services, web applications, and distributed systems. The .NET framework includes ASP.NET for web development, ADO.NET for data access, and Windows Communication Foundation (WCF) for service-oriented applications, among others.
- 4. Security:** Security was a major focus in the creation of .NET. The framework incorporates robust security mechanisms, including code access security (CAS) and validation and verification. These features help developers build secure applications and protect against common vulnerabilities.
- 5. Platform Support:** While initially targeted at Windows, the goal was also to support multiple platforms. This vision has been realized with the introduction of .NET Core (now consolidated into .NET 5 and beyond), which supports development and deployment on Linux and macOS, in addition to Windows.
- 6. Support for Cloud and Mobile:** As cloud computing and mobile development became more prevalent, .NET evolved to include support for these paradigms. .NET Core and Xamarin (for mobile app development) are examples of how .NET has adapted to these trends.

| Overview of .NET Framework release history ^{[4][5][6][7]} | | | | | | | | | |
|--|-----|----------------------------|---------------------------|---------------------------|----------------------------|---------------------|--|-------------------------------------|---------------------|
| Version | CLR | Release date | Support ended | Visual Studio | Included in | | Can be installed on | | Replaces |
| | | | | | Windows | Windows Server | Windows | Windows Server | |
| 1.0 | 1.0 | 2002-01-15 | 2009-07-14 ^[8] | Visual Studio .NET (2002) | N/A | N/A | NT 4.0 SP6a, 98, 98SE, Me, 2000, XP | NT 4.0 SP6a, 2000, 2003 | N/A |
| 1.0 SP1 | ↑ | 2002-03-19 | ↑ | | N/A | N/A | ↑ | ↑ | ↑ |
| 1.0 SP2 | ↑ | 2002-08-07 | ↑ | | XP SP1 ^[a] | N/A | ↑ | ↑ | ↑ |
| 1.0 SP3 | ↑ | 2004-08-30 ^[9] | ↑ | | N/A | N/A | ↑ | ↑ | ↑ |
| 1.1 | 1.1 | 2003-04-09 | 2013-10-09 ^[8] | Visual Studio .NET 2003 | N/A | 2003 (x86) | NT 4.0 SP6a, 98, 98SE, Me, 2000, XP, Vista | NT 4.0 SP6a, 2000, 2003 (x64), 2008 | 1.0 ^[10] |
| 1.1 SP1 | ↑ | 2004-08-30 ^[9] | ↑ | | XP SP2, SP3 ^[b] | 2003 SP1, SP2 (x86) | ↑ | ↑ | ↑ |
| 2.0 | 2.0 | 2005-10-27 ^[11] | 2011-07-12 ^[8] | Visual Studio 2005 | N/A | 2003 R2 | 98, 98SE, Me, 2000 SP3, XP SP2 | 2000 SP3, 2003 | N/A |
| 2.0 SP1 | ↑ | 2007-11-19 ^[12] | ↑ | | N/A | 2008 | 2000 SP4, XP SP2 | 2000 SP4, 2003 SP1 | ↑ |
| 2.0 SP2 | ↑ | 2008-08-11 ^[13] | ↑ | | N/A | 2008 SP2, 2008 R2 | ↑ | ↑ | ↑ |
| 3.0 | 2.0 | 2006-11-06 ^[14] | 2011-07-12 ^[8] | Visual Studio 2008 | Vista | N/A | XP SP2 | 2003 SP1 | 2.0 |

3. How we can run application in .NET?



3.1. Stage 1: Compile Time

The first phase converts source code to an intermediate representation:

1. **Source Code:** The developer writes code in a .NET language (C#, VB.NET, or another .NET language).
2. **Language-Specific Compiler:** The appropriate compiler (C# compiler, VB.NET compiler, etc.) processes the source code.
3. **Common Intermediate Language (CIL):** The compiler produces CIL code (formerly called MSIL or IL), along with metadata. This is platform-independent bytecode that contains:
 - Instructions in an intermediate form
 - Metadata about types, members, and references
 - The assembly manifest (version information, security information, etc.)

This CIL code is stored in assemblies (.dll or .exe files) and is not yet machine code. It's a universal intermediate representation that can run on any platform with a .NET runtime.

3.2. Stage 2: Runtime

The second phase converts the intermediate representation to machine code and executes it:

1. **Just-In-Time (JIT) Compilation:** When the program runs, the .NET runtime's JIT compiler converts the CIL code to native machine code specific to the current CPU architecture and operating system.
2. **Native/Machine Code Execution:** This optimized, platform-specific code is then directly executed by the CPU.

3.3. Key Advantages of This Approach

- **Platform Independence:** The same CIL code can run on any platform with a .NET runtime.
- **Performance Optimization:** The JIT compiler can optimize for the specific machine it's running on.
- **Language Interoperability:** Different .NET languages compile to the same CIL, enabling seamless integration.
- **Security and Verification:** The runtime can verify and enforce security before execution.
- In .NET Core and .NET 5+, there's also **Ahead-Of-Time (AOT) compilation**, which allows pre-compiling IL into native code before execution.

3.6. Key Differences Between Java and C#

- Java bytecode can be interpreted and JIT-compiled dynamically.
- C# IL is always JIT-compiled before execution (or AOT-compiled in some cases).
- This makes C# typically execute **faster** at startup since the CLR does not rely on interpretation.

4. Different versions of .NET

The .NET ecosystem has evolved significantly over the years, leading to the development of various branches: .NET Framework, .NET Core, and the unified .NET platform.

4.1. Why There Are So Many Versions of .NET

The proliferation of .NET versions stems from several key factors:

1. **Evolving Technology Landscape:** As computing shifted from desktop-centric to web, mobile, and cloud platforms, Microsoft needed to adapt the framework to remain competitive.
2. **Technical Debt:** The original .NET Framework, designed in the early 2000s, accumulated architectural limitations that were difficult to overcome without a fresh start.
3. **Cross-Platform Imperative:** Microsoft's strategic shift toward embracing open-source and cross-platform development necessitated a reimagined framework (.NET Core).
4. **Market Competition:** Competing with newer platforms like Node.js required performance improvements and modern features impossible to implement in the legacy framework.
5. **Acquisition and Integration:** As Microsoft acquired technologies like Xamarin, they needed to create coherence across diverse platforms.

4.2. .NET Framework

- **Launched:** Early 2000s
- **Purpose:** Originally created to support the development of applications and services on Windows.
- **Key Features:** Provides a comprehensive class library and runtime environment for building Windows desktop applications and web services. It includes technologies like Windows Forms, WPF (Windows Presentation Foundation), ASP.NET Web Forms, and ADO.NET.
- **Limitations:** It's primarily Windows-focused and does not support cross-platform development. Over time, it became evident that a more flexible, modern solution was needed to keep up with the changing landscape of software development.
- **Current Status:** Now in maintenance mode, receiving only security updates and bug fixes, with no new feature development.

4.3. .NET Core

- **Launched:** 2016
- **Purpose:** Developed as a cross-platform, open-source rewrite of the .NET Framework, .NET Core was designed to support the development of applications for Windows, Linux, and macOS.
- **Key Features:** It emphasized modularity, performance, and cross-platform support. It includes ASP.NET Core for web applications, Entity Framework Core for data access, and the ability to run on multiple operating systems. Provides excellent support for containerization (Docker) and cloud-native application development.
- **Evolution:** .NET Core was a significant shift towards modernizing the .NET ecosystem, making it more competitive and versatile in a cloud-centric, cross-platform world.

4.4. .NET (5 and onwards)

- **Launched:** .NET 5 was released in November 2020, marking the unification of the .NET platform.
- **Purpose:** Unifies the .NET Core and Xamarin platforms into a single framework with consistent APIs, runtime behaviors, and developer tools. The goal was to simplify the .NET ecosystem, making it easier for developers to choose and work with the .NET technologies.
- **Versioning Note:** The version number jumped from .NET Core 3.1 directly to .NET 5 (skipping 4) to avoid confusion with .NET Framework 4.x.
- **Key Features:** Offers a single runtime and framework that can be used everywhere, with high performance, modern language features, and support for cloud, IoT, and AI-based applications. It continues to support all the application types covered by .NET Core and adds new improvements and capabilities.
- **Future Direction:** With the release of .NET 5 and beyond (e.g., .NET 6, .NET 7, .NET 8), Microsoft has moved to an annual release schedule, providing a clear path for developers to modernize applications and take advantage of the latest features in the ecosystem.

4.5. .NET Standard

- **Purpose:** A formal specification of .NET APIs that are intended to be available across all .NET implementations.
- **Function:** Establishes uniformity in the .NET ecosystem, making it easier to develop applications and libraries that are compatible across different .NET platforms, such as .NET Framework, .NET Core, and Xamarin.
- **Benefits:** By targeting .NET Standard, developers ensure that their code can run on any platform that implements the standard, facilitating code sharing and reducing the complexity of developing cross-platform applications.
- **Current Relevance:** While crucial during the transition period between .NET Framework and .NET Core, its importance is diminishing as the unified .NET (5+) becomes the standard approach for new development.

5. Elements of the .NET Ecosystem

Common Type System (CTS): The CTS is part of the .NET architecture that defines all possible data types and programming constructs supported by the runtime. It ensures that objects written in different .NET languages can interact with each other. While the CTS is more about the runtime's type safety and interoperability features, it underpins the uniformity that .NET Standard aims to achieve across implementations.

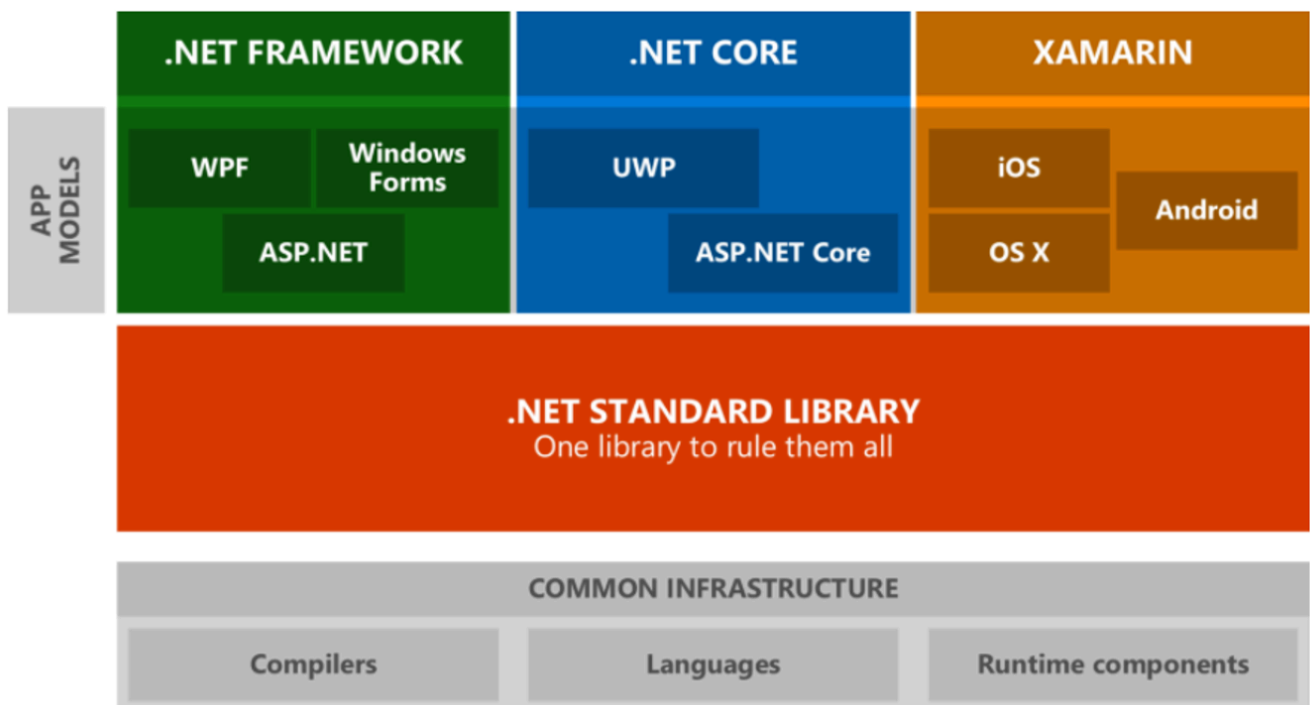
Common Language Runtime (CLR): The CLR is the execution environment for .NET applications, providing services like memory management, type safety, exception handling, and more. Each .NET implementation has its own version of the CLR (e.g., CoreCLR for .NET Core, CLR for .NET Framework). The CLR works in conjunction with the Just-In-Time (JIT) compiler to translate IL code into native machine code at runtime, optimizing performance for the specific platform.

Framework Class Library (FCL): The FCL is a large collection of reusable classes, interfaces, and value types that applications can use for common operations like I/O, threading, collections, and more. The .NET Standard specifies a subset of these libraries that should be available across all .NET implementations, ensuring code portability. In unified .NET (5+), the FCL has been streamlined and optimized for better cross-platform performance.

Metadata and Assemblies: .NET uses assemblies as the building blocks of applications, which contain metadata about the types defined in the program, including their definitions and implementations. The metadata ensures that the information about every type and member is available at runtime, facilitating features like reflection. This metadata-driven approach enables powerful tooling, dynamic loading, and inspection capabilities.

Base Class Libraries (BCL)

Part of the larger FCL, the BCL provides the most fundamental classes, including those for collections, I/O, threading, and basic types like System.String and System.DateTime. The .NET Standard includes these core libraries to ensure basic functionality is uniformly available. In modern .NET (5+), these libraries have been unified across all platforms, reducing the fragmentation that existed between different .NET implementations.



6. C# Language

C# (pronounced "C-sharp") is a modern, object-oriented, and type-safe programming language developed by Microsoft as part of its .NET initiative. It was created by Anders Hejlsberg and his team and was officially announced in 2000. The first version of C# was released in 2002 along with the .NET Framework 1.0.

C# was designed to be a simple, modern, general-purpose, object-oriented programming language, borrowing key concepts from several other languages, notably Java, C++, and Visual Basic. It was intended to provide a balance between simplicity and power, making it suitable for developing a wide range of applications.

C# has evolved significantly since its inception, with modern versions (C# 8, 9, 10, 11, and beyond) introducing powerful features like pattern matching, record types, nullable reference types, top-level statements, and improved asynchronous programming capabilities, cementing its position as one of the most versatile programming languages in use today.

7. C# Applications

1. **Desktop Applications:** C# can be used to create Windows client applications using Windows Forms (WinForms), Windows Presentation Foundation (WPF), and Universal Windows Platform (UWP). The newer MAUI (Multi-platform App UI) framework extends this capability to macOS.
2. **Web Applications:** ASP.NET Core allows developers to build high-performance, cross-platform web applications, APIs, and microservices. Blazor enables building interactive web UIs using C# instead of JavaScript.
3. **Game Development:** With frameworks like Unity, C# has become a popular choice for game developers looking to create games for consoles, PC, and mobile devices. It powers many of the world's most successful games across multiple platforms.
4. **Mobile Applications:** Using .NET MAUI (evolution of Xamarin), developers can create mobile applications for Android, iOS, and Windows with a single codebase in C#, sharing up to 90% of the code across platforms.
5. **Cloud Applications:** C# is extensively used in cloud computing, including developing and managing cloud services, serverless functions, web APIs, and more, especially on Microsoft's Azure platform. It integrates seamlessly with Azure services.
6. **Enterprise Applications:** Due to its robustness, security features, and integration with .NET libraries, C# is widely used in large-scale enterprise environments for developing complex business applications and microservices architecture.
7. **IoT and Embedded Systems:** With .NET IoT libraries, C# can be used to program IoT devices including Raspberry Pi, enabling sophisticated device control and sensor integration.
8. **AI and Machine Learning:** Using ML.NET and integrations with other ML frameworks, C# is becoming increasingly popular for building and deploying machine learning models in production environments.

8. C# - history

8.1. C# 1.0 (2002)

- **Initial Release:** Included basic features like classes, structs, interfaces, events, properties, methods, delegates, expressions, statements, and attributes.
- **Core Design:** Established C#'s philosophy of type safety, object-orientation, and component-oriented programming.

8.2. C# 2.0 (2005)

8.2.1. Generics

Create type-safe collections without boxing/unboxing performance penalties.

```
// Instead of ArrayList with boxing/unboxing
List<int> numbers = new List<int>();
numbers.Add(10); // No boxing occurs
```

8.2.2. Partial types

Partial types allow splitting a class across multiple files, useful for separating generated code from custom code or organizing large classes.

```
// File: Person.cs
public partial class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

// File: Person.Methods.cs
public partial class Person
{
    // Method in a separate file
    public string GetFullName()
    {
        return $"{FirstName} {LastName}";
    }
}
```

8.2.3. Anonymous methods

Anonymous methods let you define inline event handlers or delegates without creating separate named methods.

```
// Before anonymous methods
Button button = new Button();
button.Click += new EventHandler(button_Click);

void button_Click(object sender, EventArgs e)
{
    MessageBox.Show("Button clicked");
}

// With anonymous methods
Button button = new Button();
button.Click += delegate(object sender, EventArgs e)
{
    MessageBox.Show("Button clicked");
};
```

8.2.4. Nullable types

Nullable types allow value types like `int`, `double`, `bool` to hold null values, representing absence of data.

```
// Without nullable types
int age = GetAge(); // Throws if no age available
if (ageAvailable)
{
    int age = GetAge();
}

// With nullable types
int? age = GetAgeOrNull();
if (age.HasValue)
{
    Console.WriteLine($"Age: {age.Value}");
}
else
{
    Console.WriteLine("Age unknown");
}

// Null coalescing
int displayAge = age ?? 0; // Use 0 if age is null
```

8.2.5. Iterators

Iterators simplify creating enumerable sequences with `yield return`, letting you generate values on-demand without building complex collection classes.

```
// Without iterators - complex implementation
public class NumberCollection : IEnumerable<int>
{
    private readonly int _max;

    public NumberCollection(int max) => _max = max;

    public IEnumerator<int> GetEnumerator()
    {
        return new NumberEnumerator(_max);
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    // Requires custom enumerator class
    private class NumberEnumerator : IEnumerator<int> { /*...*/ }
}

// With iterators - simple implementation
public IEnumerable<int> GetNumbers(int max)
{
    for (int i = 0; i < max; i++)
    {
        if (i > 10 && ShouldStop())
            yield break; // Exit the enumeration early

        yield return i; // Return this value, pause, resume later
    }
}

// Usage
foreach (var num in GetNumbers(20))
{
    Console.WriteLine(num);
}
```

8.2.6. Static Classes

Static classes are ideal for utility methods that don't require object state, preventing accidental instantiation and ensuring all members are static.

```
// Static class cannot be instantiated
public static class MathUtilities
{
    // Must contain only static members
    public static double CalculateCircleArea(double radius)
    {
        return Math.PI * radius * radius;
    }

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

// Usage
double area = MathUtilities.CalculateCircleArea(5);

// Error: Cannot create instance of static class
// var utils = new MathUtilities(); // Compilation error
```

8.3.C# 3.0 (2007)

8.3.1. Auto-implemented properties

Auto-implemented properties provide a simplified syntax for property declarations when no additional logic is needed in the accessors.

```
// Before auto-implemented properties
private string _name;
public string Name
{
    get { return _name; }
    set { _name = value; }
}

// With auto-implemented properties
public string Name { get; set; }

// With initialization (C# 6.0+)
public string Name { get; set; } = "Default";

// Read-only auto-property (C# 6.0+)
public string Id { get; } = Guid.NewGuid().ToString();
```

8.3.2. Anonymous types

Anonymous types let you create simple classes on the fly without defining them explicitly, primarily used in LINQ queries.

```
// Creating an anonymous type
var person = new { Name = "John", Age = 30 };

// Accessing properties
Console.WriteLine($"{person.Name} is {person.Age} years old");

// Useful in LINQ projections
var people = new List<User>();
var result = people.Select(p => new { p.Name, YearOfBirth = DateTime.Now.Year - p.Age });
```

8.3.3. Query expressions

LINQ query expressions provide SQL-like syntax for querying collections and other data sources.

```
// Query syntax for filtering and projection
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;

// More complex query with ordering and grouping
var users = GetUsers();
var usersByCountry = from user in users
                    where user.Age >= 18
                    orderby user.LastName
                    group user by user.Country into countryGroup
                    select new {
                        Country = countryGroup.Key,
                        Count = countryGroup.Count()
                    };
};
```

8.3.4. Lambda expressions

Lambda expressions provide a concise syntax for writing anonymous functions, often used with LINQ methods.

```
// Pre-lambda with anonymous method
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
List<int> evenNumbers = numbers.FindAll(delegate(int n) { return n % 2 == 0; });

// With lambda expression
List<int> evenNumbers = numbers.FindAll(n => n % 2 == 0);

// Multi-parameter lambda
Func<int, int, int> add = (a, b) => a + b;
Console.WriteLine(add(3, 5)); // 8

// Expression with statement block
Func<int, int> factorial = n => {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
};
```

8.3.5. Extension methods

Extension methods allow adding methods to existing types without modifying or inheriting from them.

```
// Defining extension methods
public static class StringExtensions
{
    public static bool IsValidEmail(this string input)
    {
        return !string.IsNullOrEmpty(input) && input.Contains("@") && input.Contains(".");
    }

    public static string Truncate(this string input, int maxLength)
    {
        if (string.IsNullOrEmpty(input)) return input;
        return input.Length <= maxLength ? input : input.Substring(0, maxLength) + "...";
    }
}

// Using extension methods
string email = "user@example.com";
bool isValid = email.IsValidEmail();

string longText = "This is a very long text that needs truncation";
string truncated = longText.Truncate(20); // "This is a very long..."
```

8.3.6. Expression trees

Expression trees represent code as data that can be examined and manipulated at runtime, enabling dynamic query generation and compilation.

```
// Creating an expression tree for a lambda
Expression<Func<int, bool>> isPositive = num => num > 0;

// Examining the expression tree
// This represents: num > 0
// Parameter: num
// Body: (num > 0)
// Left: num
// Operator: >
// Right: 0
BinaryExpression body = (BinaryExpression)isPositive.Body;
ParameterExpression param = (ParameterExpression)body.Left;
ConstantExpression constant = (ConstantExpression)body.Right;

Console.WriteLine($"Parameter: {param.Name}");
Console.WriteLine($"Operator: {body.NodeType}");
```

```

Console.WriteLine($"Constant: {constant.Value}");

// Compiling and using an expression tree
Func<int, bool> compiled = isPositive.Compile();
bool result = compiled(5); // true

```

8.3.7. Implicitly typed local variables

The `var` keyword allows the compiler to infer the type of a variable from its initializer, reducing redundancy while maintaining strong typing.

```

// Without var - type repeated
Dictionary<string, List<Customer>> customersByRegion = new Dictionary<string, List<Customer>>();

// With var - cleaner but still strongly typed
var customersByRegion = new Dictionary<string, List<Customer>>();

// The compiler infers appropriate types
var name = "John"; // string
var age = 30; // int
var isActive = true; // bool
var now = DateTime.Now; // DateTime

// Especially useful with anonymous types
var person = new { Name = "Alice", Age = 25 }; // No explicit type available

```

8.3.8. Object and collection initializers

Object and collection initializers provide concise syntax for setting properties and adding items during instantiation.

```

// Before object initializers
Customer customer = new Customer();
customer.Name = "John Smith";
customer.Email = "john@example.com";
customer.Age = 35;

// With object initializer
Customer customer = new Customer
{
    Name = "John Smith",
    Email = "john@example.com",
    Age = 35
};

// Before collection initializers
List<string> colors = new List<string>();
colors.Add("Red");
colors.Add("Green");
colors.Add("Blue");

// With collection initializer
List<string> colors = new List<string> { "Red", "Green", "Blue" };

// Combining object and collection initializers
Customer customer = new Customer
{
    Name = "John Smith",
    Email = "john@example.com",
    Orders = new List<Order>
    {
        new Order { Id = 1, Amount = 100 },
        new Order { Id = 2, Amount = 200 }
    }
};

```

8.5.C# 4.0 (2010)

8.4.1. Dynamic binding

Dynamic binding defers type checking from compile-time to runtime, enabling easier interaction with dynamic languages, COM objects, and reflection.

```
// Using dynamic type
dynamic obj = GetSomeObject(); // Could be anything at runtime

// No compile-time checking - resolved at runtime
obj.SomeMethod(123);
string result = obj.SomeProperty;

// Example with Excel COM interop
dynamic excel = Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));
excel.Visible = true;
excel.Workbooks.Add();
excel.Cells[1, 1].Value = "Hello World";
```

8.4.2. Named and optional arguments

Named arguments allow specifying arguments by parameter name, while optional arguments can be omitted when the method defines default values.

```
// Method with optional parameters
public void DisplayMessage(string message, bool isError = false, ConsoleColor color = ConsoleColor.White)
{
    Console.ForegroundColor = isError ? ConsoleColor.Red : color;
    Console.WriteLine(message);
    Console.ResetColor();
}

// Calling with positional arguments
DisplayMessage("An error occurred", true);

// Calling with named arguments - can be in any order
DisplayMessage(
    isError: true,
    message: "An error occurred"
);

// Skipping optional parameters
DisplayMessage("Normal message"); // Uses defaults for isError and color

// Specifying only some optional parameters
DisplayMessage("Warning", color: ConsoleColor.Yellow); // Skips isError
```

8.4.3. Generic co- and contravariance

Covariance and contravariance enable more flexible type relationships with generic interfaces and delegates, allowing subtypes in output positions and supertypes in input positions.

```
// Covariance - allows more derived type in output position
// IEnumerable<T> is covariant in T (defined with 'out' keyword)
IEnumerable<string> strings = new List<string>();
IEnumerable<object> objects = strings; // Valid with covariance

// Contravariance - allows less derived type in input position
// Action<T> is contravariant in T (defined with 'in' keyword)
Action<object> objectAction = obj => Console.WriteLine(obj);
Action<string> stringAction = objectAction; // Valid with contravariance

// Custom covariant interface
```

```

interface IProducer<out T>
{
    T Produce(); // T only in output position
}

// Custom contravariant interface
interface IConsumer<in T>
{
    void Consume(T item); // T only in input position
}

```

8.4.4. Embedded interop types

Embedded interop types ("NoPIA") eliminate the need for Primary Interop Assemblies when interacting with COM components by embedding only the necessary type information directly into your assembly.

```

// Before: Required explicit reference to interop assembly (PIA)
// References: Microsoft.Office.Interop.Excel.dll

// With embedded interop types: Add reference with "Embed Interop Types" = true
// Then use normally, but no PIA needed at runtime
using Microsoft.Office.Interop.Excel;

public void CreateExcelSheet()
{
    Application excel = new Application();
    excel.Visible = true;
    Workbook workbook = excel.Workbooks.Add();
    Worksheet sheet = workbook.ActiveSheet;

    // Type info is embedded in your assembly
    // No need to distribute Interop DLL
    sheet.Cells[1, 1].Value = "Hello World";
}

```

8.5.C# 5.0 (2012)

8.5.1. Asynchronous members

Async and await keywords simplify asynchronous programming by allowing you to write asynchronous code that looks like synchronous code, without complex callbacks or state machines.

```
// Before async/await - complex callback approach
public void DownloadWebpageOld(string url)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += (sender, e) => {
        if (e.Error != null)
            HandleError(e.Error);
        else
            ProcessContent(e.Result);
    };
    client.DownloadStringAsync(new Uri(url));
}

// With async/await - linear, easier to read and maintain
public async Task DownloadWebpageAsync(string url)
{
    try
    {
        HttpClient client = new HttpClient();
        string content = await client.GetStringAsync(url);
        ProcessContent(content);
    }
    catch (Exception ex)
    {
        HandleError(ex);
    }
}

// Multiple async operations in sequence
public async Task<UserData> GetUserDataAsync(int userId)
{
    // Each await yields control back to the caller while waiting
    var user = await _userRepository.GetUserAsync(userId);
    var orders = await _orderRepository.GetOrdersForUserAsync(userId);
    var preferences = await _preferencesService.GetPreferencesAsync(userId);

    return new UserData
    {
        User = user,
        Orders = orders,
        Preferences = preferences
    };
}
```

8.5.2. Caller information attributes

Caller information attributes automatically populate method parameters with information about the calling code, useful for logging, debugging, and diagnostics.

```
public void LogMessage(
    string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    Console.WriteLine($"Message: {message}");
    Console.WriteLine($"Member: {memberName}");
    Console.WriteLine($"File: {sourceFilePath}");
}
```

```
        Console.WriteLine($"Line: {sourceLineNumber}");
    }

    // When called, the compiler automatically provides the caller info
    public void ProcessOrder()
    {
        LogMessage("Processing order");
        // Compiler transforms this to:
        // LogMessage("Processing order", "ProcessOrder", "C:\\Project\\OrderService.cs", 42);
    }
}
```

8.6.C# 6.0 (2015)

8.6.1. String interpolation

String interpolation provides a more readable and concise syntax for formatting strings by embedding expressions directly in string literals.

```
// Before string interpolation - using String.Format
string name = "John";
int age = 30;
string message = String.Format("My name is {0} and I am {1} years old.", name, age);

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

// Can include expressions
string message = $"My name is {name} and I'll be {age + 1} next year.";

// Can format numbers and dates
double price = 123.45;
DateTime now = DateTime.Now;
string formatted = $"Price: {price:C2} on {now:d}"; // $123.45 on 3/13/2025
```

8.6.2. Expression-bodied function members

Expression-bodied members provide a concise syntax for methods, properties, and other members that consist of a single expression.

```
// Before - traditional method
public double CalculateArea(double radius)
{
    return Math.PI * radius * radius;
}

// With expression body
public double CalculateArea(double radius) => Math.PI * radius * radius;

// Property with expression body
public string FullName => $"{FirstName} {LastName}";

// Get-only property with expression body
public bool IsAdult => Age >= 18;

// In C# 7.0+, also works for constructors, destructors, etc.
public Person(string name) => Name = name;
```

8.6.3. Null-conditional operators

Null-conditional operators (`?.` and `?[]`) provide a concise way to access members or elements only when the target is not null, returning null instead of throwing exceptions.

```
// Before null-conditional operators
string name = null;
int? length = null;
if (name != null)
{
    length = name.Length;
}

// With null-conditional operator
string name = null;
int? length = name?.Length; // Returns null instead of throwing

// Chaining with null-conditional operators
```

```
int? firstZipCode = customer?.Address?.ZipCode;

// Null-conditional with indexers
string firstItem = items?[0]; // Null if items is null

// Combining with null-coalescing
string displayName = person?.Name ?? "Unknown";

// With method calls
string upper = name?.ToUpper(); // Null if name is null
```

8.6.4. Auto-property initializers

Auto-property initializers allow setting a default value for an auto-implemented property directly at declaration.

```
// Before auto-property initializers
public class Product
{
    private string _category;
    public string Category { get; set; }

    public Product()
    {
        _category = "General";
    }
}

// With auto-property initializers
public class Product
{
    public string Category { get; set; } = "General";
    public DateTime CreatedDate { get; set; } = DateTime.Now;
    public List<string> Tags { get; set; } = new List<string>();

    // Can also be used with read-only properties
    public string Id { get; } = Guid.NewGuid().ToString();
}
```

8.6.5. Exception filters

Exception filters allow specifying a condition for an exception handler to execute, without unwinding the stack if the condition is false.

```
// Before exception filters
try
{
    ProcessFile(path);
}
catch (IOException ex)
{
    // Catch, then check condition inside
    if (ex.HResult == -2147024784) // File not found
    {
        // Handle file not found
    }
    else
    {
        // Need to re-throw if it's not the specific error we want to handle
        throw;
    }
}

// With exception filters
try
{
    ProcessFile(path);
}
```

```

}
catch (IOException ex) when (ex.HResult == -2147024784) // File not found
{
    // Only executes for file not found errors
    Console.WriteLine("File not found, creating it...");
    File.Create(path);
}
catch (IOException ex) when (IsRecoverable(ex))
{
    // Can even call methods in the filter condition
    AttemptRecovery();
}
}

```

8.6.6. Nameof expressions

The `nameof` expression returns the name of a variable, type, or member as a string constant, providing compile-time checking for refactoring and preventing magic strings.

```

// Before nameof
public void ProcessOrder(string customerId)
{
    if (string.IsNullOrEmpty(customerId))
    {
        // Magic string - breaks if parameter name changes
        throw new ArgumentNullException("customerId");
    }
}

// With nameof
public void ProcessOrder(string customerId)
{
    if (string.IsNullOrEmpty(customerId))
    {
        // Compiler-checked - will update if parameter name changes
        throw new ArgumentNullException(nameof(customerId));
    }
}

// Works with properties
public string Name { get; set; }
public void ValidateName()
{
    logger.Log($"Validating {nameof(Name)}");
}

// Works with types
Console.WriteLine(nameof(DateTime)); // "DateTime"

```

8.6.7. Index initializers

Index initializers provide a concise syntax for initializing dictionaries and other indexed collections.

```

// Before index initializers
var states = new Dictionary<string, string>();
states.Add("CA", "California");
states.Add("NY", "New York");
states.Add("TX", "Texas");

// With index initializers
var states = new Dictionary<string, string>
{
    ["CA"] = "California",
    ["NY"] = "New York",
    ["TX"] = "Texas"
};

```

```
// Can be combined with collection initializers
var states = new Dictionary<string, string>
{
    // Collection initializer syntax
    { "WA", "Washington" },
    { "OR", "Oregon" },

    // Index initializer syntax
    ["CA"] = "California",
    ["NY"] = "New York"
};
```

8.6.8. Await in catch/finally blocks

Support for await in catch and finally blocks enables proper asynchronous cleanup and error handling.

```
public async Task ProcessFileAsync(string path)
{
    Resource resource = null;
    try
    {
        resource = await OpenResourceAsync(path);
        await ProcessResourceAsync(resource);
    }
    catch (Exception ex)
    {
        // Can now use await inside catch blocks
        await LogExceptionAsync(ex);

        // Can await different recovery strategies
        if (ex is FileNotFoundException)
            await TryCreateFileAsync(path);
        else
            await NotifyAdminAsync(ex);
    }
    finally
    {
        // Can now use await for cleanup in finally
        if (resource != null)
            await resource.CloseAsync();
    }
}
```

8.7.C# 7.0-7.3 (2017-2018)

8.7.1. Tuples and deconstruction

Tuples provide a lightweight syntax for grouping multiple values without defining a specific type, while deconstruction allows extracting those values into separate variables.

```
// Before tuples - using custom class or out parameters
public class NameParts { public string First; public string Last; }
public NameParts SplitName(string fullName) { /*...*/ }

// Using tuples to return multiple values
public (string First, string Last) SplitName(string fullName)
{
    var parts = fullName.Split(' ');
    return (parts[0], parts[1]);
}

// Calling and using the tuple return value
var nameParts = SplitName("John Doe");
Console.WriteLine($"First: {nameParts.First}, Last: {nameParts.Last}");

// Deconstruction - extracting tuple elements to separate variables
(string firstName, string lastName) = SplitName("John Doe");
Console.WriteLine($"First: {firstName}, Last: {lastName}");

// Simplified deconstruction with var
var (first, last) = SplitName("John Doe");

// Deconstruction works with existing types that define Deconstruct method
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) => (x, y) = (X, Y);
}

var point = new Point(10, 20);
var (x, y) = point; // Calls Deconstruct method
```

8.7.2. Pattern matching

Pattern matching extends the capabilities of type checking with the ability to test for conditions and extract values in a single operation.

```
// Before pattern matching
public void ProcessValue(object value)
{
    if (value is int)
    {
        int number = (int)value;
        if (number > 0)
            ProcessPositiveNumber(number);
    }
    else if (value is string)
    {
        string text = (string)value;
        if (!string.IsNullOrEmpty(text))
            ProcessText(text);
    }
}
```

```
// With pattern matching – type patterns with conditions
public void ProcessValue(object value)
{
    // Type pattern with declaration
    if (value is int number)
    {
        // number is already cast and available here
        if (number > 0)
            ProcessPositiveNumber(number);
    }
    // Type pattern with condition
    else if (value is string text && !string.IsNullOrEmpty(text))
    {
        ProcessText(text);
    }
}

// Switch pattern matching
public string Classify(object item)
{
    switch (item)
    {
        case null:
            return "Unknown";
        case int n when n < 0:
            return "Negative Number";
        case int n when n > 0:
            return "Positive Number";
        case string s when s.Length > 0:
            return "Non-Empty String";
        case string s:
            return "Empty String";
        case Person p when p.Age < 18:
            return "Minor";
        case Person p:
            return "Adult";
        default:
            return "Something else";
    }
}
}
```

8.7.3. Local functions

Local functions allow defining methods inside other methods, limiting their scope and capturing local variables.

```
public int Calculate(int[] values)
{
    // Local function defined inside another method
    int Square(int n) => n * n;

    int sum = 0;
    foreach (var value in values)
    {
        // Call the local function
        sum += Square(value);
    }

    return sum;
}

// Local function with access to outer variables
public void ProcessItems(IEnumerable<Item> items)
{
    var errorCount = 0;

    // Local function that captures the errorCount variable
    void LogError(string message)
    {

```

```

        Console.WriteLine($"Error: {message}");
        errorCount++;
    }

    foreach (var item in items)
    {
        if (!item.IsValid)
            LogError($"Invalid item: {item.Id}");
    }

    Console.WriteLine($"Processed with {errorCount} errors");
}

// Local functions can be called before they're defined (unlike lambdas)
public bool IsPrime(int number)
{
    if (number <= 1) return false;

    return !HasDivisors(number);

    // Defined after it's used
    bool HasDivisors(int n)
    {
        for (int i = 2; i <= Math.Sqrt(n); i++)
        {
            if (n % i == 0) return true;
        }
        return false;
    }
}

```

8.7.4. Ref locals and returns

Ref locals and returns allow methods to return references to variables rather than copies, enabling more efficient operations on large value types.

```

// Without ref returns – returns a copy
public Point FindPoint(Point[] points, Predicate<Point> match)
{
    for (int i = 0; i < points.Length; i++)
    {
        if (match(points[i]))
            return points[i]; // Returns a copy
    }
    return default;
}

// With ref returns – returns a reference
public ref Point FindPointRef(Point[] points, Predicate<Point> match)
{
    for (int i = 0; i < points.Length; i++)
    {
        if (match(points[i]))
            return ref points[i]; // Returns a reference
    }
    throw new InvalidOperationException("No matching element found");
}

// Using ref locals and returns
Point[] points = new Point[100];
// Initialize points...

// Get reference to the element
ref Point foundPoint = ref FindPointRef(points, p => p.X > 10);

// Modify directly in the array through the reference
foundPoint.X = 100; // Changes the value in the array

```

```
// Finding largest value by reference (more efficient for large structs)
public ref int FindLargest(int[] numbers)
{
    ref int largest = ref numbers[0];

    for (int i = 1; i < numbers.Length; i++)
    {
        if (numbers[i] > largest)
            largest = ref numbers[i];
    }

    return ref largest;
}
```

8.7.5. Out variables

Out variables simplify the use of out parameters by declaring the variables directly at the method call site.

```
// Before out variables
int number;
if (int.TryParse("123", out number))
{
    Console.WriteLine($"Parsed: {number}");
}

// With out variables – inline declaration
if (int.TryParse("123", out int number))
{
    Console.WriteLine($"Parsed: {number}");
}

// Out variables with var
if (int.TryParse("123", out var result))
{
    Console.WriteLine($"Parsed: {result}");
}

// Multiple out parameters
public void GetCoordinates(out int x, out int y) => (x, y) = (10, 20);

// Using multiple out variables
GetCoordinates(out var x, out var y);
Console.WriteLine($"X: {x}, Y: {y}");
```

8.7.6. Discards

Discards are placeholder variables (using the `_` character) that intentionally ignore values, used when you don't care about certain return values or parameters.

```
// Using discards with tuples when you only care about some values
var (firstName, _, age) = GetPersonInfo(); // Ignore the middle value

// Discards with deconstruction
var point = GetPoint();
var (x, _) = point; // Only care about the x coordinate

// Discards with out parameters
// Before – had to declare variables we don't use
if (int.TryParse("123", out int number)) { }

// With discards – clearly shows we're ignoring the result
if (int.TryParse("123", out _)) { }

// Multiple discards in pattern matching
if (GetValue() is var (_, _, z))
```

```

{
    // Only use z
}

// Discards in switch expressions (C# 8.0+)
string category = item switch
{
    Person(_, _, var age) when age < 18 => "Minor",
    _ => "Unknown" // Discard pattern for default case
};

```

8.7.7. Span<T> and Memory<T>

Span<T> and Memory<T> provide high-performance, allocation-free access to contiguous memory regions, ideal for processing large arrays or working with native memory.

```

using System;

// Using Span for slicing arrays without allocation
byte[] array = new byte[100];
Span<byte> firstHalf = array.AsSpan(0, 50);
Span<byte> secondHalf = array.AsSpan(50, 50);

// Fill each half with different values
firstHalf.Fill(1);
secondHalf.Fill(2);

// Span works with stackalloc for stack-allocated memory
Span<byte> stackSpan = stackalloc byte[128]; // Allocated on stack, no heap allocation
stackSpan[0] = 100;
stackSpan[127] = 200;

// Memory<T> is similar but can be stored in fields (unlike Span)
public void ProcessLargeArray(int[] data)
{
    // Process chunks of the array without creating copies
    for (int i = 0; i < data.Length; i += 1000)
    {
        int length = Math.Min(1000, data.Length - i);
        Span<int> chunk = data.AsSpan(i, length);
        ProcessChunk(chunk);
    }
}

// High-performance string manipulation without allocations
public bool StartsWithHello(ReadOnlySpan<char> text)
{
    ReadOnlySpan<char> hello = "Hello".AsSpan();
    return text.StartsWith(hello, StringComparison.OrdinalIgnoreCase);
}

```

8.7.8. Non-trailing named arguments

Named arguments can now appear before positional arguments, providing more flexibility in method calls.

```

// Method with multiple parameters
public void DisplayMessage(string message, bool isError, ConsoleColor color, int priority)
{
    // Implementation...
}

// Before C# 7.2: Named arguments had to appear after all positional arguments
DisplayMessage("Error occurred", true, color: ConsoleColor.Red, priority: 1);

// With C# 7.2+: Named arguments can appear anywhere
DisplayMessage(message: "Error occurred", true, color: ConsoleColor.Red, 1);

```

```
// Particularly useful for readability with boolean parameters
ProcessOrder(id: 1234, rushOrder: true, false, amount: 100.50);

// Helps with readability when some parameters have obvious values
SendEmail(
    to: "user@example.com",
    "Important notification", // subject is obvious from context
    sendCopy: false,
    "Your account has been updated" // body is obvious from context
);
```

8.7.9. Improved overload resolution

C# 7.2 and 7.3 enhanced the compiler's ability to select the most appropriate method overload, particularly with generics and optional parameters.

```
// Generic constrained method overloads
public void Process<T>(T value) where T : class { }
public void Process<T>(T value) where T : struct { }

// Before improvement: Ambiguous call
// Process("hello"); // Compiler error

// With improvement: Compiler selects class constraint version
Process("hello"); // Works in C# 7.3+

// Method with optional parameters
public void Display(int value, bool highlight = false) { }
public void Display(string value, bool highlight = false) { }

// Better overload resolution with literals
Display(42); // Selects int overload
Display("42"); // Selects string overload

// Field targets for auto-properties (C# 7.3)
public class Person
{
    private string _firstName; // Field created explicitly

    // Now can target this field with auto-property
    public string FirstName
    {
        get => _firstName;
        set => _firstName = value?.Trim();
    }
}
```

8.8. C# 8.0 (2019)

- **Nullable reference types:** Introduced syntax for indicating whether a reference type is expected to be null.
- **Asynchronous streams:** Supported asynchronous iteration using `await foreach`.
- **Default interface methods:** Allowed interfaces to define default implementations for members.
- **Pattern matching enhancements:** Extended pattern matching capabilities with switch expressions.
- **Indices and ranges:** Introduced new syntax for accessing elements from the end of a collection and for slicing collections.
- **Using declarations:** Simplified resource management with a more concise syntax.
- **Static local functions:** Guaranteed that local functions don't capture (reference) any variables.
- **Readonly members:** Allowed struct methods to specify they don't modify state.
- **Null-coalescing assignment:** Simplified assigning a value when the variable is null.

8.9. C# 9.0 (2020)

- **Records:** Provided a concise syntax for creating immutable reference types with value-based equality.
- **Init only setters:** Allowed properties to be made immutable after object initialization.
- **Top-level statements:** Simplified entry point syntax for applications by removing boilerplate.
- **Pattern matching enhancements:** Further improved pattern matching with logical patterns.
- **Target-typed new expressions:** Allowed omission of the type when instantiating an object if the type is already known.
- **Covariant return types:** Enabled override methods to return more specific types.
- **Extension GetEnumerator:** Allowed foreach to be used with extension methods.
- **Lambda discard parameters:** Improved lambda expressions with discards for unused parameters.
- **Native sized integers:** Added support for platform-dependent integer sizes with `nint` and `nuint`.

8.10. C# 10.0 (2021)

- **Record structs:** Introduced record types for structs, combining value semantics with record features.
- **Global using directives:** Enabled global application of using directives across all source files in a project.
- **File-scoped namespace declaration:** Simplified namespace declaration to reduce nesting.
- **Extended property patterns:** Enhanced pattern matching with properties.
- **Constant interpolated strings:** Allowed interpolated strings to be used as constants.
- **Improvements to structs:** Added parameterless constructors and field initializers to structs.
- **Lambda expression improvements:** Added natural syntax for lambda return types.
- **Improved definite assignment:** Enhanced the compiler's ability to track variable assignments.
- **CallerArgumentExpression attribute:** Enabled capturing the expressions passed to methods.

8.11. C# 11.0 (2022)

- **List patterns:** Added support for matching sequences in a more concise way.
- **Required members:** Introduced a way to specify that a member must be initialized when an object is created.
- **Checked and Unchecked expressions:** Allowed expressions to specify checked or unchecked context.
- **Static abstract members in interfaces:** Enabled definition of static abstract methods in interfaces.
- **UTF-8 string literals:** Added support for UTF-8 string literals with the `u8` suffix.
- **Raw string literals:** Introduced multi-line strings without escape sequences.
- **Generic attributes:** Allowed attributes to use generic types.
- **Auto-default structs:** Improved struct constructors to automatically initialize fields.
- **Pattern matching on `ReadOnlySpan<char>`:** Extended pattern matching to work with spans.

8.12. C# 12.0 (2023)

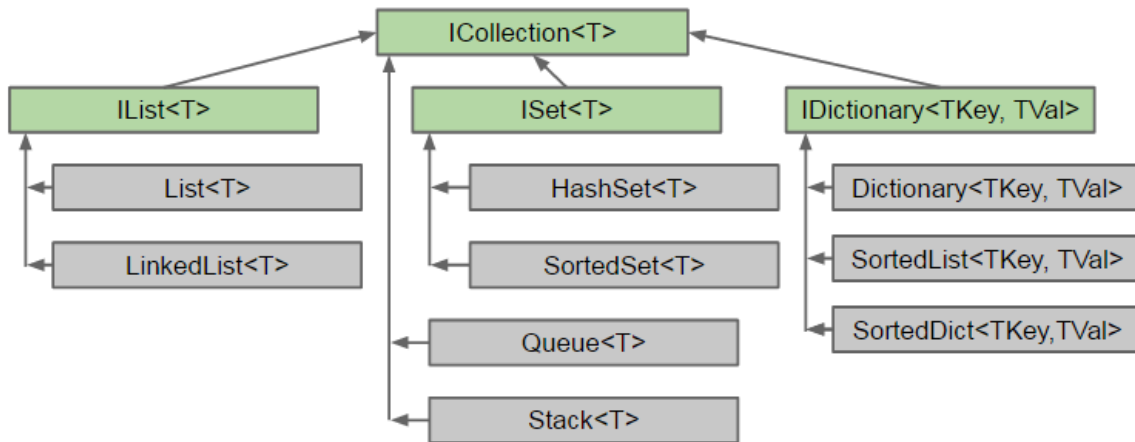
- **Primary constructors:** Simplified class and struct declarations with constructor parameters directly in the type declaration.
- **Collection expressions:** Provided a concise syntax for initializing collections.
- **Inline arrays:** Allowed for creation of array-like types that allocate on the stack.
- **Optional parameters in lambda expressions:** Enabled lambda expressions with optional parameters.
- **Ref readonly parameters:** Added support for passing read-only references as parameters.
- **Alias any type:** Extended using directives to create aliases for any type, including tuples and arrays.

- **Experimental attribute:** Marked APIs as experimental, warning about potential future changes.
- **Interceptors:** Enabled method call interception for advanced scenarios (preview feature).

8.13. C# 13.0 (2024)

- **Enhanced collection expressions:** Extended collection expressions to support more collection types and complex scenarios.
- **Type parameter defaults:** Simplified generic type declarations with default values for type parameters.
- **Using field parameters:** Reduced boilerplate for field initialization via constructor parameters.
- **Semi-auto properties:** Introduced properties with default getters but custom setters.
- **Inline scopes:** Allowed scoping variables to expressions instead of blocks.
- **Advanced pattern matching:** Added support for nested patterns and more complex matching scenarios.
- **Null-checking directives:** Enabled project-wide null checking behavior customization.
- **Discriminated unions:** Provided first-class support for sum types (preview feature).

9. Collections



9.1. IEnumerable<T>

x

9.2. ICollection<T> (Base Interface)

The root interface for generic collections that defines basic collection operations.

9.3. Main Collection Types

9.3.1. Lists (IList<T>)

Ordered collections allowing duplicates and indexed access.

```
// List<T> - Dynamic array with resizing
List<string> names = new List<string> { "Alice", "Bob", "Charlie" };
names.Add("David");
string secondName = names[1]; // "Bob"
names.Remove("Alice");

// LinkedList<T> - Doubly-linked list
LinkedList<int> numbers = new LinkedList<int>();
numbers.AddLast(10);
numbers.AddFirst(5);
numbers.AddAfter(numbers.First, 7);
foreach (var num in numbers) { } // 5, 7, 10
```

9.3.2. Sets (ISet<T>)

Collections with unique elements.

```
// HashSet<T> - Unordered unique elements, fast lookups
HashSet<string> uniqueNames = new HashSet<string> { "Alice", "Bob" };
uniqueNames.Add("Alice"); // Won't add duplicate
bool contains = uniqueNames.Contains("Bob"); // O(1) lookup

// SortedSet<T> - Ordered unique elements
SortedSet<int> sortedNumbers = new SortedSet<int> { 5, 3, 8, 1 };
foreach (var num in sortedNumbers) { } // 1, 3, 5, 8
```

9.3.3. Dictionaries (IDictionary<TKey, TValue>)

Key-value pair collections.

```
// Dictionary<TKey, TValue> - Fast key lookups
Dictionary<string, int> ages = new Dictionary<string, int>
{
    ["Alice"] = 30,
    ["Bob"] = 25
};
ages.Add("Charlie", 35);
int bobAge = ages["Bob"]; // 25

// SortedDictionary<TKey, TValue> - Keys in sorted order
SortedDictionary<string, double> prices = new SortedDictionary<string, double>
{
    ["Banana"] = 1.29,
    ["Apple"] = 0.99
};
// Iterates in alphabetical order: Apple, Banana

// SortedList<TKey, TValue> - Sorted by key, less memory than SortedDictionary
SortedList<int, string> ranked = new SortedList<int, string>
{
    [3] = "Bronze",
    [1] = "Gold",
    [2] = "Silver"
};
// Iterates in key order: 1:Gold, 2:Silver, 3:Bronze
```

9.3.4. Stack and Queue

Stack and queue data structures.

```
// Stack<T> - LIFO (Last In, First Out)
Stack<string> stack = new Stack<string>();
stack.Push("First");
stack.Push("Second");
string item = stack.Pop(); // "Second"

// Queue<T> - FIFO (First In, First Out)
Queue<string> queue = new Queue<string>();
queue.Enqueue("First");
queue.Enqueue("Second");
string item = queue.Dequeue(); // "First"
```

9.3.5. Summary

Each collection type has specific performance characteristics and use cases. The choice depends on your needs for ordering, lookup speed, insertion/deletion patterns, and whether you need unique elements.